University of Zurich[UZH]

Department of Informatics

Scalable Graph Processing With Signal/Collect

Dissertation submitted to the Faculty of Business,
Economics and Informatics
of the University of Zurich

to obtain the degree of
Doktor der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by
Philip Stutz
from Embrach ZH

approved in July 2015

at the request of
Prof. Dr. Abraham Bernstein and
Prof. Dr. Axel Polleres

University of Zurich[UZH]

The Faculty of Business, Economics and Informatics of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, July 15, 2015

Chairwoman of the Doctoral Board: Prof. Dr. Elaine M. Huang

Abstract

Our ability to process large amounts of data and the size and number of data sets are growing at an incredible pace. This development presents us with the opportunity to build systems that perform complex analyses of increasingly dense networks of data. These opportunities include computing recommendations, analysing social networks, finding patterns in transaction networks, scheduling tasks, or inferencing on probabilistic models. Many of these tasks involve processing data that has a natural graph representation.

Whilst the opportunities are there in the form of access to processing resources and data sets, the way we write software has largely not caught up. Many use MapReduce for scalable processing, but this abstraction has shortcomings with regard to processing graph structured data, especially with iterative and asynchronous processing.

This thesis introduces the SIGNAL/COLLECT programming model and framework for efficient parallel and distributed large-scale graph processing. We show that this abstraction captures the essence of many algorithms on graphs in a concise and elegant way. Beyond that, we also show implementations of two complex systems built on SIGNAL/COLLECT: The first system is TripleRush, a distributed in-memory triple store with a novel architecture. The second system is foxPSL, a distributed probabilistic inferencer. Our evaluations show that the SIGNAL/COLLECT framework can efficiently execute simple graph algorithms such as PageRank. They also show that the two complex systems have competitive performance relative to the respective state-of-the-art.

For this reason we believe that SIGNAL/COLLECT is suitable for designing scalable dynamic and complex systems that process large networks of data.

Zusammenfassung

Unsere Kapazität grosse Datenmengen zu verarbeiten und die Grösse und Anzahl von Datensätzen wachsen rasant. Diese Entwicklung eröffnet uns die Möglichket Softwaresysteme zu bauen, die zusehends komplexere Analysen auf immer dichteren Netzwerken von Daten ausführen. Anwendungsgebiete dafür sind zum Beispiel Empfehlungsdienste, die Analyse sozialer Netzwerke, das Auffinden von Mustern in Transaktionen, das Optimieren von Aufgabenplanungen, oder Inferenz in probabilistischen Modellen. Viele dieser Anwendungen erfordern die Verarbeitung von Daten, die sich am natürlichsten als Graph repräsentieren lassen.

Wir haben heute zwar Zugang zu den Verabeitungsressourcen und Datensätzen, aber die Art und Weise wie wir Software schreiben hat damit nur beding Schritt gehalten. Viele verwenden MapReduce für die skalierende Datenverarbeitung, aber diese Abstraktion hat Schwächen bei der Verarbeitung von Daten mit Graph Struktur, speziell bei iterativen und asynchronen Berechnungen.

In dieser Arbeit präsentieren wir das SIGNAL/COLLECT Programmiermodell und Softwaresystem zur effizienten parallelen und verteilten Verarbeitung von grossen Graphen. Wir zeigen, dass diese Abstraktion es erlaubt viele Graph Algorithmen elegant und prägnant zu formulieren. Zusätzlich zeigen wir Implementierungen von zwei komplexen Systemen, die ebenfalls mit SIGNAL/COLLECT gebaut wurden: Das erste System ist TripleRush, ein verteilter in-memory Triple Store mit einer neuartigen Architektur. Das zweite ist foxPSL, ein verteiltes System für probabilistische Inferenz.

Unsere Evaluationen zeigen, dass SIGNAL/COLLECT einfache Graph Algorithmen wie zum Beispiel PageRank effizient berechnen kann. Sie zeigen ebenfalls, dass die zwei komplexen Systeme von der Leistung her mit den jeweiligen modernsten Systemen mithalten können.

Aus diesem Grund glauben wir, dass SIGNAL/COLLECT für das Design von skalierenden dynamischen und komplexen Systemen geeignet ist, die grosse Netzwerke von Daten verarbeiten.

Acknowledgements

I thank my adviser and mentor Dr. Abraham Bernstein for having given me the opportunity to do research in his group. I appreciate that he gave me the freedom to build experimental systems and I admire his ability to quickly structure problems and systematically address them. It was a privilege to be able to learn from him these past years. I also thank co-advisor Dr. Axel Polleres for his valuable feedback. I am very grateful to the Hasler Foundation and the University of Zurich for having funded my research.

The remaining acknowledgements are more personal: Thank you, Ela, for being a great friend. It was both inspiring and rewarding to work with you and you made our office a friendly and cheerful place. Thank you, Sara, for the collaboration on PSL. I'm proud of what we built and your ambition carried that project. I thank all the students I had the opportunity to work with, especially Daniel, Stefan, and Carol. Thank you, Lorenz, for being a great friend, for your dedication when redoing Inf4Oek together, and for your reliable righteousness in general. Thanks, Esther, for initially getting me in touch with Avi. I thank the DDIS & IfI cosmos that I spent time with: Adrian, Amancio, Andi, Avi, Barbara, Bibek, Cosmin, Christian, Daniel Spicar, Daniel Strebel, Doro, Ela, Flory, Helen, HP, Iris, Jayalath, Jonas, Juk, Kathi, Khoa, Lorenz, Marc, Mike, Patrick de Boer, Patrick Minder, Ping, Shen, Tobi, and Tom. You all made it a pleasure to come to IfI, I really appreciate our conversations and coffee breaks. I also thank my family and friends for being there for me, especially my dad for his support and for his encouragement to work on things I feel passionate about.

Table of Contents

I Synopsis

Introduction 3

Contribution Summaries 4

Structure and Relationships 7

Limitations and Future Work 11

Conclusion 16

II Publications

Signal/Collect 24
Philip Stutz, Daniel Strebel, and Abraham Bernstein

Distributed TripleRush 82
Philip Stutz, Bibek Paudel, Mihaela Verman, and Abraham Bernstein

foxPSL 103
Sara Magliacane, Philip Stutz, Paul Groth, and Abraham Bernstein

Part I

Synopsis

Introduction

Our ability to store, communicate and compute information is growing exponentially [13]. Today we have access to commodified clusters and many large data sets. These developments present us with the opportunity to build systems that perform complex analyses of increasingly dense networks of data. These opportunities include computing recommendations, analysing social networks, finding patterns in transaction networks, scheduling tasks, or inference in probabilistic models. Many of these tasks involve processing data that has a natural graph representation.

Building custom parallel and distributed systems is hard and time consuming, which is why many developers use specialised programming models that transparently parallelise and distribute computations that fit the model. Many use MapReduce [7], but its model is often not a natural fit for graph structured data [6, 21] or for expressing algorithms that have iterative components [5, 8, 31, 32, 1]. For this reason, researchers have explored specialised programming models that are more suitable for processing large graphs and capable of expressing iterative algorithms [15, 23, 28, 22, 19]. Finding a better programming model for processing complex networks of data is also the goal of this thesis. More specifically, the goal was to design a programming model that can be used to both express simple graph algorithms, as well as scalable dynamic and complex systems that process large networks of data.

Contribution Summaries

The goal of this thesis is to design and evaluate a programming model for building systems that process complex networks of data. We pursued this goal by first designing the programming model and framework with a focus on its evaluation on simple algorithms and graph analytics tasks. Then we designed two complex systems with different requirements in that programming model, in order to demonstrate that the model is also suitable for building complex systems. In the following we outline the individual contributions.

Signal/Collect (described in Part II, Chapter 1) is our proposed graph processing model and framework. The contributions are:

- *Design of an expressive programming model for parallel and distributed computations on graphs.* The model supports many features such as:
  - decomposition of the computation into state updates and communication
  - multiple vertex/edge types in the same graph
  - asynchronous scheduling
  - local and global convergence detection
  - dynamic graph modifications
  - pluggable heuristics for vertex placement (configurable trade-off between load-balancing and locality)
  - suitability for computations in which data is iteratively updated
  - suitability for computations in which data flows through a network of processing vertices

  We demonstrate the expressiveness of the model by giving implementations of algorithms from categories as varied as clustering, ranking, classification, and constraint optimisation.
- *We built, evaluated, and released a framework for this model.* The framework efficiently and scalably parallelises and distributes algorithms expressed in the programming model. We empirically show that our framework scales to multiple cores, with increasing dataset size, and in a distributed setting. We evaluated real-world scalability

by computing PageRank on the Yahoo! AltaVista webgraph,[1] where the framework is competitive with PowerGraph [11].
- *We evaluated the impact of asynchronous algorithm executions.* SIGNAL/COLLECT supports both synchronous and asynchronous algorithm executions. We compare the difference in runtimes between the asynchronous and synchronous execution mode for different algorithms and problems of varying complexity.

TripleRush (described in Part II, Chapter 2) is a high-performance parallel and distributed in-memory triple store. The architecture was implemented on top of SIGNAL/COLLECT. The contributions are:

- *We designed and released a novel distributed triple store architecture* in which query processing is implemented as a parallel exploration of an index structure that is optimised for efficiently routing (partial) query descriptions.
- *We evaluated the scalability*. In our two scalability benchmarks, the architecture shows good vertical-, horizontal-, and data-scalability.
- *We evaluated the time until first results*. For some queries the architecture was able to return first results in as little as ~30% of the time until full results, which is made possible by the asynchronous design.
- *We benchmarked the performance and scalability against two other single-node triple stores on two datasets*. In these benchmarks TripleRush has better data scalability and performance than both Sesame and Virtuoso. On the largest evaluated data-sets TripleRush is between 10 and 200 times faster than the other evaluated systems.
- *We benchmarked the distributed performance at billion triples scale*. The performance measured for TripleRush is competitive with the numbers reported for other systems.

foxPSL (described in Part II, Chapter 3) is a language and system for expressing and efficiently solving probabilistic inferencing problems. The language is an extension of Probabilistic Soft Logic (PSL) [4, 18, 2, 3]. The distributed inferencing is built on SIGNAL/COLLECT. The contributions are:

- *We designed and released an end-to-end PSL environment that supports distributed inference.*

[1] Yahoo! Academic Relations, Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`

- *The system seamlessly parallelises and distributes computations.* In our evaluations it outperformed the state of the art both in inference time and in terms of solution quality.
- *We support a broad range of extended PSL features and optimisations.*

Next we place SIGNAL/COLLECT and the the two systems in the overall structure of this thesis.

Structure and Relationships

In the previous chapter we described the individual contributions of our programming model and framework as well as the ones of two system use cases. In this chapter we embed these components into the overall structure of the thesis and we point out the relationships between them.

Figure 1 gives an overview of the different use cases for SIGNAL/COLLECT and connects them with the relevant parts of the thesis. The foundation is the SIGNAL/COLLECT programming model and framework, the basis of both the simple and system use cases. The simple use cases are part of the programming model evaluation described in Section 5 of Chapter 1 in Part II. TripleRush and foxPSL are the two larger system use cases that form part of this thesis which are described in detail in Part II, Chapters 2 and 3 respectively. There are more system use cases that are not part of this thesis: one is a framework for easily expressing and extending local iterative algorithms for distributed constraint optimisation problems [30] and the other one is a system for detecting patterns of fraudulent financial transactions [27]. In this chapter we focus on the two systems that are part of this thesis.

Why these two systems? We wanted to demonstrate that SIGNAL/COLLECT allows to express scalable dynamic and complex systems that process large networks of data. The way to show this is by building such systems. The two systems we built are suitable for this, because they have very different requirements and together use most features of the programming model. This is illustrated in Table 1 which compares some of the relevant system properties and used SIGNAL/COLLECT features.

TripleRush has two fundamentally different aspects modelled in SIGNAL/COLLECT: On the one hand there is the multitree structured index, with different vertex implementations for different levels. On the other hand there is the query vertex, which is the starting point of a query execution, coordinates query optimisation, determines when query execution has finished, and reports the results. The index does not do a lot of computation, but depending on the data represented by the index there can a be a lot of branching when sending messages down the multitree. The query vertex is dynamically added when a query is passed to TripleRush

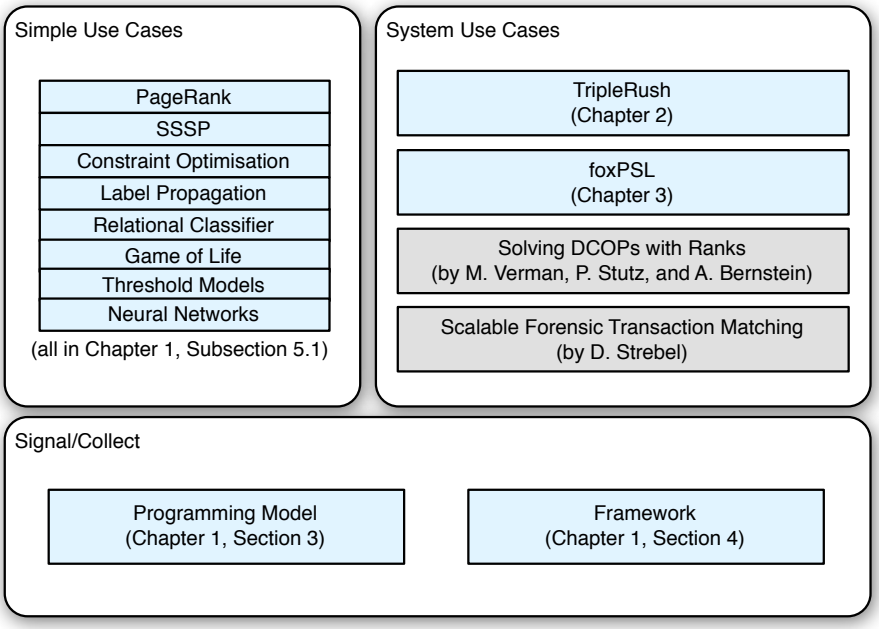


Fig. 1. Overview of the SIGNAL/COLLECT model, system, and use cases. Entries with blue backgrounds are part of this thesis and include a reference to the relevant chapter.

and removed after the results have been reported. The signals that are routed down the index structure contain information that allows to potentially route them back into the multitree during the query execution, or, if successful, back to the query vertex.

TripleRush is messaging-heavy, which is why it makes use of several combiners in addition to bulk messaging in order to reduce the number and size of messages that are exchanged. The TripleRush execution is completely asynchronous, which allows to return first results before the full result has been computed. The asynchronous execution also avoids dependencies between unrelated parts of the execution, which reduces the latency compared to an execution with additional synchronisations.

foxPSL has signals iterating back and forth in a bipartite graph, updating the states of the involved vertices until convergence. All edges are bidirectional, modelled as two directed edges in SIGNAL/COLLECT. The synchronous execution allows for all these computations to proceed

in lock-step without additional synchronisation messages. Consequently, we can assign meaning to not sending a message, which is why foxPSL only sends messages when something has changed. If no message was received, a vertex assumes that the missing signals were the same as the last ones that were received along those edges. This is why for foxPSL a synchronous execution is more efficient than an asynchronous execution.

The vertices in one half of the bipartite graph run computationally expensive convex optimisations, which SIGNAL/COLLECT transparently parallelises and distributes over multiple computers. In the future it might make sense to try to also use heuristics to improve the messaging locality by placing more vertices on the same computer as their neighbours in the bipartite graph.

Another feature offered by SIGNAL/COLLECT that foxPSL currently does not use yet, but that could be useful, are dynamic graph modifications (when a new fact is added, the whole inference currently has to be re-run). Dynamic graph modifications would allow us to only incrementally recompute the parts of the graph that are affected by the change. foxPSL uses local convergence detection to stop computations locally when the changes are below a given threshold. It also uses global convergence detection, implemented with MapReduce-style aggregations, to determine if the global error is small enough to stop the inference.

Both systems take advantage of optimisations in the framework such as efficient edge representations[2] or reducing the number of sent messages with bulk messaging. But more generally, both systems share that it was possible to express them in the SIGNAL/COLLECT programming model. This means that TripleRush and foxPSL are both system architectures that can be decomposed into a huge network of small pieces of data, functions that compute messages that are exchanged between the pieces of data (signal functions), and functions that can update a piece of data based on received messages (collect functions). This decomposition is in both cases unique to the data and to the problem that is being solved, but fundamentally it is this decomposition that enables SIGNAL/COLLECT to efficiently execute those systems. We believe that being able to express two such different systems in the SIGNAL/COLLECT model in-

[2] By using compressed integer sets to represent all IDs of the vertices that the edges point to. Implementation at `https://github.com/uzh/signal-collect/blob/master/src/main/scala/com/signalcollect/util/SplayIntSet.scala`

dicates that we reached the goal of designing a programming model for building systems that process complex networks of data.

System Comparison	**TripleRush**	**foxPSL**
Number of vertex types	4-12 (some are very similar)	2
Execution mode	asynchronous	sync. (async. less efficient)
Processing style	data-flow	data-graph
Graph structure	multitree (explicit index edges)	bipartite
Fraction of vertices involved	small (per query)	all
Messaging load/structure	query/data dependent	rule/fact dependent
Computation in vertex	low	high
Uses combiners	yes	no
Uses bulk messaging	yes	yes
Dynamic graph modifications	yes	not yet (could be useful)
Local convergence detection	no	yes
Global convergence detection	no	yes
Distributed placement heuristics	yes	not yet (could be useful)
Compact edge representation	yes	yes

Table 1. Comparison of TripleRush and foxPSL with regard to their implementation in SIGNAL/-COLLECT.

In the next chapter we look at the limitations and interesting future work in the context of designing systems with SIGNAL/COLLECT.

Limitations and Future Work

In this chapter we look at the limitations of SIGNAL/COLLECT and we identify interesting future work that might address them.

SIGNAL/COLLECT is both a programming model and a framework and we will individually look at limitations of the current implementation, as well as at limitations of the programming model.

Framework and efficient execution

Many limitations of the framework are due to a lack of engineering resources, examples include the absence of error recovery or cloud deployment options. In this section we would like to focus on more fundamental limitations of how algorithms expressed in the SIGNAL/COLLECT model are executed.

In-Memory The current implementation only works in-memory and it is an interesting question if an efficient implementation of an on-disk version akin to GraphChi [20] for the PowerGraph [11] model is possible. The advantage would be the ability to scale to larger datasets with a smaller memory footprint or on a single machine; the downside would be slower executions due to higher latency and lower throughput.

Earlier explorations by Daniel Strebel based on key-value stores [26] indicate that for SIGNAL/COLLECT using external storage incurs a huge performance penalty. One way to address this might be to change the internals, which are currently based on hash maps, to instead optimise for sequential operations. This would lead to fewer high-latency seeks on disk drives and would probably also increase throughput and the effectiveness of caches in general. Changing the internals to rely on mostly sequential operations would most likely require other trade-offs: TripleRush, for example, needs to be able to route messages to vertices to which there is no explicitly stored edge. It is an open question if this feature and others, such as dynamic graph modifications, could be implemented efficiently with predominantly sequential storage operations.

Alternative computation architectures The SIGNAL/COLLECT framework can currently only execute algorithms on CPUs. Alternative archi-

tectures such as GPUs hold the promise of superior parallel performance, but developers have to adapt their programs to some additional architectural constraints. Existing research for doing graph processing on GPUs suggests that dynamic graph modifications might be difficult to support and memory constraints might also require adaptations to how operations are scheduled [10], but that in some cases huge speedups of more than 100 are possible relative to pure CPU architectures [9].

Manual optimisations Although one can execute both TripleRush and foxPSL with the standard framework facilities, reaching competitive performance required to use optimised components, for example for the memory efficient storage of edges. Ideally a user should be able to specify the algorithm in a language that can be analysed by the execution environment in order to perform such optimisations automatically. Other optimisations can maybe not be performed statically, but the execution environment could analyse the computation and perform adaptive optimisations at runtime. Green-Marl [14], for example, defines a domain specific language for graph processing, where the user defined computations can be optimised for the execution platform. It is an interesting question to what degree one can create algorithm representations that allow for automated optimisation, whilst preserving the expressiveness and flexibility required to build complex systems.

Partitioning SIGNAL/COLLECT uses random hash-based partitioning by default, but allows to plug in a custom function for mapping vertex IDs to workers and nodes. TripleRush, for example, uses such a mapping in order to ensure that many child vertices in the multitree end up on the same node as their parent. The random partitioning results in an even distribution of the vertices over the workers and nodes, which is good from a load-balancing point of view. The problem is that the random partitioning is almost guaranteed to have bad locality: When there are many nodes, then most edges will traverse a node boundary, which leads to higher latency and expensive serialisation.

There are different approaches to tackling this: PowerGraph partitions the edges, which can be used to achieve better load balancing and allows to smear the load of high in-/out-degree vertices across workers and machines. The downside is that it requires that the sender side knows about the *gather* function used by the target vertex. Translated

to the SIGNAL/COLLECT model that would mean that each edge would need to know about the *collect* function of the receiving vertex, which would add significant overhead in the case of heterogeneous graphs with different vertex types. In addition to this, having outgoing edges for one vertex in different places would require synchronising the state of that vertex, which would likely add latency or constrain the scheduling.

Thomas Keller experimented with other approaches to achieve a better graph partitioning for SIGNAL/COLLECT [16]. Before running algorithms he partitioned the graphs into balanced partitions that have few edges between them. By assigning vertex IDs that get mapped to the desired partition by the default hash-based mapping function, one can avoid the memory overhead of storing explicitly what partition each vertex belongs to. In his explorations [16] the overhead of partitioning was rather large, compared to the performance gains during algorithm execution, which is consistent with other reported results [24]. Explorations that did the repartitioning dynamically during algorithm execution faced both the issue of having the partitioning overhead during the measured execution time, as well as the memory overhead of maintaining a potentially large table that stores what partition each vertex belongs to. The external memory based graph processing system Mizan [17] shows good results for dynamic repartitioning and addresses the partition/routing table issue by using a distributed hash table. It is an interesting question if this solution would also work well for an in-memory graph processing system, where additional latency due to indirect routing might be more noticeable.

Christian Tschanz analysed static partitioning for TripleRush [29], because as a long-running system it is more likely to be able to amortise the partitioning overhead, compared to short-running algorithms such as PageRank. The idea was to analyse both the index structure and a number of executed queries, and then repartition the index based on this information. This approach did deliver performance gains on the queries that were directly optimised, but it is unclear how well these gains would translate to queries that are different, but still similar to the ones that were used for the optimisation.

Programming model

Flexibility vs. Complexity The SIGNAL/COLLECT model offers many features, such as asynchronous execution and dynamic modifications. With this flexibility combined with a vertex-centric perspective, which is new to most developers, it can be a challenge to navigate all this complexity when debugging a distributed system. We tried to address this with inspectable visualisations of the graph and by allowing to test and observe programs locally during interactive synchronous step-by-step executions.[3] In spite of this, there is still a lot of potential for better tools when it comes to diagnosing issues in a system based on the SIGNAL/-COLLECT programming model.

Model too low-level In SIGNAL/COLLECT there are issues that a developer needs to manage, which could plausibly be simplified: For example it is not possible for the framework to transparently replicate processing vertices, such as the ones in the TripleRush index. For this reason a developer would have to manually manage this and create multiple instances. Some other structures, for example the per-query convergence detection mechanism in TripleRush, seem like they are patterns that should be reusable. Might we be able to build systems with an even higher-level model, where whole subgraphs could be instantiated and reused according to structural rules? There is ongoing work towards building such higher-level constructs for graph processing [25].

Model too high-level Compared to low-level programming models, SIGNAL/COLLECT allows to express algorithms in a way that does not require a developer to explicitly send messages or decide on which node a computation happens. This makes expressing many algorithms convenient, but some aspects of the model seem to make too many assumptions and constrain the freedom of the developer. For example, the model requires for each vertex to have a state that is updated. TripleRush uses an optimisation offered by the framework (but not the model) that allows it to fuse collecting and signalling and not modify the state. This means that underneath the SIGNAL/COLLECT programming model there is the framework model, which does support more flexibility, but only at the

[3] Interactive execution mode: `https://github.com/uzh/signal-collect/wiki/Execution-modes`

cost of being more complex. This suggests that there might be a fundamental trade-off between the potential for optimisations and complexity, and it is unclear if shifting the model to a higher level is necessarily the right answer, if building high-performance systems still requires more flexibility and lower-level optimisations.

For which systems is SIGNAL/COLLECT *the right abstraction?* With TripleRush and foxPSL we have shown that SIGNAL/COLLECT can support building vastly different systems. But this does not yet answer the question for which systems SIGNAL/COLLECT is the best solution. We do have some intuitions, for example if the system can be decomposed into a complex network of small code and data pieces that interact with each other only via messages. But this description would also match the actor model [12]. When is SIGNAL/COLLECT the better choice than modelling something with actors? We are unsure. A simplified answer could focus on the various features of the model, such as if the communication structure should be accessible and potentially modified during the execution, or if the system requires synchronisation or convergence detection that can be naturally expressed in SIGNAL/COLLECT. Another aspect is scale: SIGNAL/COLLECT addresses many issues that one would run into when designing an actor system in which billions of actors communicate with each other. Our intuition is that the answer is also connected to graphs: If a graph seems like the right abstraction to model a system around, then SIGNAL/COLLECT can be understood as an expressive model for building and modifying this system. In the end it probably boils down to SIGNAL/COLLECT being a higher-level extension of the actor model, which comes both with convenient features that allow for a simpler and more compact description of some graph-structured systems, but also imposes some additional constraints.

Conclusion

The main contribution of this thesis is the SIGNAL/COLLECT programming abstraction that allows programmers to implement algorithms without worrying about the specifics of how the computation is executed on parallel and distributed processing resources. The proof of concept systems have demonstrated that SIGNAL/COLLECT allows to build complex systems that process large amounts of graph structured data, and that those systems have performance characteristics that are competitive with the state-of-the-art.

In terms of impact, Google Scholar counts more than 70 citations for the original version of the SIGNAL/COLLECT paper that was published at ISWC. TripleRush, whilst having a novel index structure and excellent evaluation results, so far has not seen the level of attention that we believe it deserves. The work on foxPSL is still too recent to gauge its impact. The source code of all described systems has been open sourced on the code sharing and collaboration platform GitHub.[4] More than one hundred developers have shown interest in the implementation of SIGNAL/COLLECT by starring[5] it and there is an implementation of a programming model that was inspired by SIGNAL/COLLECT.[6] Part of the interest stems from a talk at the open source conference FOSDEM,[7] which has resulted in code contributions from a healthcare analytics company that was actively using the project.

We believe that SIGNAL/COLLECT—understood as a synthesis of the actor model with a graph abstraction—is useful beyond simple graph algorithms and that it is more generally useful for building scalable dynamic and complex systems that process large networks of data.

[4] `https://github.com/uzh/signal-collect`, `https://github.com/uzh/triplerush`, `https://github.com/uzh/fox`

[5] Starring is a way to show appreciation and bookmark projects on GitHub.

[6] `https://github.com/thi-ng/fabric`

[7] `https://archive.fosdem.org/2013/schedule/event/signal_collect/`

References

[1] Foto N. Afrati, Magdalena Balazinska, Anish Das Sarma, Bill Howe, Semih Salihoglu, and Jeffrey D. Ullman. Designing good algorithms for mapreduce and beyond. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 26:1–26:2, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1761-0. doi: 10.1145/2391229.2391255. URL `http://doi.acm.org/10.1145/2391229.2391255`.

[2] Stephen H. Bach, Matthias Broecheler, Lise Getoor, and Dianne P. O'Leary. Scaling MPE inference for constrained continuous Markov random fields. In *NIPS*, 2012.

[3] Stephen H. Bach, Bert Huang, Ben London, and Lise Getoor. Hinge-loss Markov random fields: Convex inference for structured prediction. In *UAI*, 2013.

[4] Matthias Broecheler, Lilyana Mihalkova, and Lise Getoor. Probabilistic similarity logic. In *UAI*, 2010.

[5] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010. ISSN 2150-8097.

[6] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29 –41, july-aug. 2009. ISSN 1521-9615. doi: 10.1109/MCSE.2009.120.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1251254.1251264`.

[8] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In Salim Hariri and Kate Keahey, editors, *HPDC*, pages 810–818. ACM, 2010. ISBN 978-1-60558-942-8.

[9] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph

analytics on gpus. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 2:1–2:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2982-8. doi: 10.1145/2621934.2621936. URL http://doi.acm.org/10.1145/2621934.2621936.

[10] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 345–354. ACM, 2012.

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, Berkeley, CA, USA, 2012. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387883.

[12] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[13] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, April 2011. ISSN 1095-9203. doi: 10.1126/science.1200970. URL http://dx.doi.org/10.1126/science.1200970.

[14] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying scalable graph processing with a domain-specific language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 208. ACM, 2014.

[15] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system. *Data Mining, IEEE International Conference on*, 0:229–238, 2009. ISSN 1550-4786.

[16] Thomas Keller. Graph partitioning for signal/collect. Master's thesis, 2013.

[17] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of*

the 8th ACM European Conference on Computer Systems, pages 169–182. ACM, 2013.

[18] Angelika Kimmig, Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *NIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012.

[19] Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, and Henri Bal. A high-level framework for distributed processing of large-scale graphs. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 155–166, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387884`.

[21] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0214-2.

[22] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[23] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *SIGMOD*. ACM, 2010. ISBN 978-1-4503-0032-2.

[24] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. Technical report, Stanford, http://infolab.stanford.edu/gps/publications/tech-report.pdf, 2012.

[25] Semih Salihoglu and Jennifer Widom. Help: High-level primitives for large-scale graph processing. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 3:1–3:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2982-8. doi: 10.1145/2621934.2621938. URL `http://doi.acm.org/10.1145/2621934.2621938`.

[26] Daniel Strebel. Making signal/collect scale. B.s. thesis, University of Zurich, 2011.

[27] Daniel Strebel. Scalable forensic transaction matching and its application for detecting patterns of fraudulent financial transactions. Master's thesis, 2013.

[28] Philip Stutz, Abraham Bernstein, and William W. Cohen. Signal/-Collect: Graph Algorithms for the (Semantic) Web. In *ISWC*, 2010.

[29] Christian Tschanz. Query-driven index partitioning for triplerush. B.s. thesis, University of Zurich, 2014.

[30] Mihaela Verman, Philip Stutz, and Abraham Bernstein. Solving distributed constraint optimization problems using ranks. In *Statistical Relational AI. Papers Presented at the Twenty-Eighth AAAI Conference on Artificial Intelligence.*, pages 125–130, Palo Alto, California, 2014. AAAI Press. ISBN 978-1-57735-674-5.

[31] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. imapreduce: A distributed computing framework for iterative computation. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1112–1121, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4577-6. doi: 10.1109/IPDPS.2011.260. URL `http://dx.doi.org/10.1109/IPDPS.2011.260`.

[32] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 13:1–13:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038929. URL `http://doi.acm.org/10.1145/2038916.2038929`.

Part II

Publications

Signal/Collect
Processing Large Graphs in Seconds

This chapter is based on an article published in the Semantic Web Journal [56],[8] which is in turn a significant extension of the ideas described in [54]. The co-authors have kindly agreed to permit inclusion as part of this thesis.

[8] http://www.semantic-web-journal.net/content/signalcollect-processing-large-graphs-seconds-1

Signal/Collect

Processing Large Graphs in Seconds

Philip Stutz, Daniel Strebel, and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
philip@stutz.tech, daniel.strebel@gmail.com,
bernstein@ifi.uzh.ch

Abstract. Both researchers and industry are confronted with the need to process increasingly large amounts of data, much of which has a natural graph representation. Some use MapReduce for scalable processing, but this abstraction is not designed for graphs and has shortcomings when it comes to both iterative and asynchronous processing, which are particularly important for graph algorithms. This paper presents the Signal/Collect programming model for scalable synchronous and asynchronous graph processing. We show that this abstraction can capture the essence of many algorithms on graphs in a concise and elegant way by giving Signal/Collect adaptations of algorithms that solve tasks as varied as clustering, inferencing, ranking, classification, constraint optimisation, and even query processing. Furthermore, we built and evaluated a parallel and distributed framework that executes algorithms in our programming model. We empirically show that our framework efficiently and scalably parallelises and distributes algorithms that are expressed in the programming model. We also show that asynchronicity can speed up execution times.
Our framework can compute a PageRank on a large (>1.4 billion vertices, >6.6 billion edges) real-world graph in 112 seconds on eight machines, which is competitive with other graph processing approaches.

1 Introduction

The Web (including the Semantic Web) is full of graphs. Hyperlinks and RDF triples, tweets and social network relationships, citation and trust networks, ratings and reviews – almost every activity on the web is most naturally represented as a graph. Graphs are versatile data structures and can be considered a generalisation of other important data structures such as lists and trees. In addition, many structures—be it physical such as transportation networks, social such as friendship networks, or virtual such as computer networks—have natural graph representations.

Graphs were at the core of the Semantic Web since its beginning. RDF, the core standard of the Semantic Web, represents a directed labeled graph. Hence, all processing of Semantic Web data includes at

least some graph processing. Initially, many systems tried to use traditional processing approaches. Triple stores, for example, tried to leverage research in relational databases to gain scalability. The most significant speed-gains, however, came from taking into account the idiosyncrasies of storing graphs [59, 44]. Another example would be the advantages gained in non-standard reasoning through the use of graphs: Learning on the Semantic Web was initially based on using traditional propositional learning methods. It was the use of statistical relational learning methods that leveraged the graph-nature of the data that allowed combining statistical and logical reasoning [30]. This combination lead to significant gains.

Coupled with the ever expanding amounts of computation and captured data [23], this means that researchers and industry are presented with the opportunity to do increasingly complex processing of growing amounts of graph structured data.

In theory, one could write a scalable program from the ground up for each graph algorithm in order to achieve the maximum amount of parallelism. In practice, however, this requires a lot of effort and is in many cases unnecessary, because many graph algorithms such as PageRank can be decomposed into small iterated computations that each operate on a vertex and its local neighbourhood (or messages from the neighbours). If we can design programming models to express this decomposition and execute the partial computations in parallel on scalable infrastructure, then we can hope to achieve scalability without having to build custom-tailored solutions.

MapReduce is the most popular scalable programming model [10], but has shortcomings with regard to iterated processing [4, 13, 62, 63] and requires clever mappings to support graph algorithms [9, 35]. Such limitations of more general programming models have motivated specialised approaches to graph processing [28, 41]. Most of these approaches follow the bulk-synchronous parallel (BSP) model [57], where a parallel algorithm is structured as a sequence of computation and communication steps that are separated by global synchronisations. The rigid pattern of bulk operations and synchronisations does not allow for flexible scheduling strategies.

To address the limitations of BSP, researchers have designed programming models for graph processing that are asynchronous [36], allow hierarchical partial synchronisations [32], make synchronisation optional

[54], or try to emulate the properties of an asynchronous computation within a synchronous model [58].

Our proposed solution is a vertex-centric programming model and associated implementation for scalable graph processing. It is designed for scaling on the commodity cluster architecture. The core idea lies in the realisation that many graph algorithms can be decomposed into two operations on a vertex: (1) signalling along edges to inform neighbours about changes in vertex state and (2) collecting the received signals to update the vertex state. Given the two core elements we call our model SIGNAL/COLLECT. The programming model supports both synchronous and asynchronous scheduling of the signal and collect operations.

Such an approach has the advantage that it can be seen as a graph extension of the actors programming approach [22]. Developers can focus on specifying the communication (i.e., graph structure) and the signalling/collecting behaviour without worrying about the specifics of resource allocation. Since SIGNAL/COLLECT allows multiple types of vertices to coexist in a graph the result is a powerful framework for developing graph-centric systems. A foundation especially suitable for Semantic Web applications, as we showed in our development of TripleRush [55] – a high-performance, in-memory triple store implemented with several different vertex types.

We extend our previous work [54] on SIGNAL/COLLECT with a more detailed description of the programming model, a larger selection of algorithm adaptations, a distributed version of the underlying framework, and with more extensive evaluations on larger graphs. Given the above, our contributions are as follows:

- *We designed an expressive programming model for parallel and distributed computations on graphs.*
  We demonstrate its expressiveness by giving implementations of algorithms from categories as varied as clustering, ranking, classification, constraint optimisation, and query processing. The programming model is also modular and composable: Different vertices and edges can be combined in the same graph and reused in different algorithms. Additionally the model supports asynchronous scheduling, data-flow computations, dynamic graph modifications, incremental recomputations, aggregation operations, and automated termination detection. Note that especially the dynamic graph modifications are

central for Web of Data applications as they require the seamless integration of ex-ante unknown data.

– *We evaluated a framework that implements the model.*
The framework efficiently and scalably parallelises and distributes algorithms expressed in the programming model. We empirically show that our framework scales to multiple cores, with increasing dataset size, and in a distributed setting. We evaluated real-world scalability by computing PageRank on the Yahoo! AltaVista webgraph.[1] The computation on the large (>1.4 billion vertices, >6.6 billion edges) graph took slightly less than two minutes using eight commodity machines, which is competitive with PowerGraph.

– *We illustrate the impact of asynchronous algorithm executions.*
SIGNAL/COLLECT supports both synchronous and asynchronous algorithm executions. We compare the difference in running times between the asynchronous and synchronous execution mode for different algorithms and problems of varying hardness.

In Section 2 we motivate the programming model and describe the basic approach. We then introduce the SIGNAL/COLLECT programming model in Section 3 and describe our implementation of the model in Section 4. In Section 5 we evaluate both the programming model and the implementation. We continue with a description of related approaches to scalable graph processing and compare them to our approach in Section 6. In Section 7 we examine the limitations of our evaluation and provide an outlook on future work. We finish by sharing our conclusions in Section 8.

2 The SIGNAL/COLLECT Approach: an Intuition

SIGNAL/COLLECT can be understood as a vertex-centric graph processing abstraction akin to Pregel [41]. Another way of looking at it is as an extension of the asynchronous actor model [22], where each vertex represents an actor and edges represent the communication structure between actors. The graph abstraction allows for the composition and evolution of

[1] Yahoo! Academic Relations, Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`

complex systems, by adding and removing vertices and edges. To illustrate this intuition we provide two examples: RDFS subclass inferencing and the computation of the single source shortest path.

RDFS Subclass Inferencing Consider a graph with RDFS classes as vertices and edges from superclasses to subclasses (i.e., `rdfs:subClassOf` triples). Every vertex has a set of superclasses as state, which initially only contains itself. Now all the superclasses send their own states as signals to their subclasses, which collect those signals by setting their own new state to the union of the old state and all signals received. It is easy to imagine how these steps, when repeatedly executed, iteratively compute the transitive closure of the `rdfs:subClassOf` relationship in the vertex states.

Single Source Shortest Path Consider a graph with vertices that represent locations and edges that represent paths between locations. We would like to determine the shortest path from a special location `S` to all the other locations in the graph.

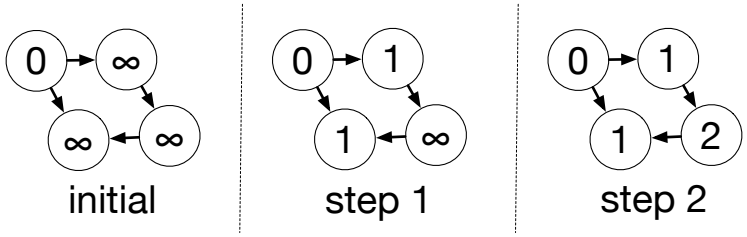


Fig. 1. States of a synchronous single-source shortest path computation with four vertices

Every location starts out with its state set to the length of the shortest currently known path from `S`. That means, initially, the state of `S` is set to 0 and the states of all the other locations are set to infinity (see Figure 1). In a first step, all edges signal the state of their source location plus the path length (represented by edge weight, in the example above all paths have length 1) to their respective target location. The target locations collect these signals by setting their new state to the lowest signal received (as long as this is smaller than their state). In a second step, the same signal/collect operations get executed using the updated vertex states. By repeating the above steps these operations

iteratively compute the lengths of the shortest paths from `S` to all the other locations in the graph.

In the next section we refine this abstraction to a programming model that allows to concisely express algorithms similar to these examples.

3 The SIGNAL/COLLECT Programming Model

In the SIGNAL/COLLECT programming model all computations are executed on a graph, where the vertices and edges both have associated data and computation. The vertices interact by the means of signals that flow along the edges. Vertices collect the signals and perform some computation on them employing, possibly, some vertex-state, and then signal their neighbours in the graph.

3.1 Basic SIGNAL/COLLECT Graph Structure

The basis for any SIGNAL/COLLECT computation is the graph

$$G = (V, E),$$

where V is the set of vertices and E the set of edges in G. During execution signals (essentially messages containing algorithm specific data items) are sent between vertices along edges.

More specifically, every *vertex* $\mathtt{v} \in V$ is a computational unit that can maintain some state and has an associated *collect* function that updates the state based on the prior state and received signals. Vertices have at least the following attributes:

- $\mathtt{v}.\mathbf{id}$, a unique id.
- $\mathtt{v}.\mathbf{state}$, the current vertex state which represents computational intermediate results. The algorithm definition needs to specify an initial state.
- $\mathtt{v}.\mathbf{outgoingEdges}$, a list of all edges $\mathtt{e} \in E$ with `e.source = v`.
- $\mathtt{v}.\mathbf{signalMap}$, a map with the ids of vertices as keys and signals as values. Every key represents the id of a neighbouring vertex and its value represents the most recently received signal from that neighbour. We use the alias `v.signals` to refer to the list of values in `v.signalMap`.

$\mathbf{v}.\mathbf{uncollectedSignals}$, a list of signals that arrived since the collect operation was last executed on this vertex.

An *edge* $\mathtt{e} \in E$ is also a computational unit directionally connecting two vertices that has an associated *signal* function that specifies what information is extracted from its source vertex and signalled to its target vertex. Hence, every *edge* $\mathtt{e} \in E$ has the following attributes:

$\mathbf{e}.\mathbf{source}$, a reference to the source vertex
$\mathbf{e}.\mathbf{target}$, a reference to the target vertex

In the most general model *signals* are messages containing algorithm specific data items. The computational model makes no assumption about the structure of signals beyond that they are computed by signal functions of the edges along which they are transmitted and processed by the collect function of the target vertex.

In a practical implementation, vertices, edges, and signals will most probably be implemented as objects. We outline such an implementation in Section 4. For this reason we also allow for additional attributes on the vertices and edges.

Example Consider data about people and family relationships between them: How could one map this to the SIGNAL/COLLECT programming model? The most direct approach is to represent each person with a vertex. If the name of a person is unique, it could be used as the id of the vertex, if not, then the name could be stored as an attribute on the vertex. The 'motherOf' relationship could then be represented as a directed edge between the vertices that represent the mother and her child.

To specify an algorithm in the SIGNAL/COLLECT programming model one needs to define the types of vertices in the compute graph with their associated `collect()` functions and the types of edges in the compute graph with their associated `signal()` functions. Note that the implementation of the `signal()` and `collect()` functions are interdependent. The `signal()` function creates a signal, which the `collect()` functions needs to be able to process. The `collect()` function in turn updates the vertex' state, which the `signal()` function needs to be able to read. These methods can both return values of arbitrary types, which means that vertex states and signals can have arbitrary types as well. To

instantiate the algorithm for execution one needs to create the actual compute graph consisting of instances of the vertices, with their initial states, and the edges. Here one needs to specify which two vertex instances are connected by an edge instance. The result is an instantiation of a graph that can be executed.

We have now defined the basic structures of the programming model. In order to completely define a SIGNAL/COLLECT computation we still need to describe how to execute computations on them.

3.2 The Computation Model and Extensions

In this section we specify how both synchronous and asynchronous computations are executed in the SIGNAL/COLLECT programming model. Also we provide extensions to the core model.

In order to precisely describe the scheduling we will need additional operations on a vertex. These operations broadly correspond to the scheduler triggering communication (doSignal) or a state update (doCollect):

```
v.doSignal()
  lastSignalState := state
  for all (e ∈ outgoingEdges) do
    e.target.uncollectedSignals.append(e.signal())
    e.target.signalMap.put(e.source.id, e.signal())
  end for

v.doCollect()
  state := collect()
  uncollectedSignals := Nil
```

The additional `lastSignalState` attribute on vertices stores the vertex state at the time of signalling, which is later used for automated convergence detection.

The `doCollect()` method updates the vertex state using the algorithm-specific `collect()` method and resets the `uncollectedSignals`. The `doSignal()` method computes the signals for all edges and relays them to the respective target vertices. With these methods we can describe a synchronous SIGNAL/COLLECT execution.

Synchronous Execution A synchronous computation is specified in Algorithm 1. Its parameter `num_iterations` defines the number of iterations (computation steps the algorithm is going to perform).

Everything inside the inner loops is executed in parallel, with a global synchronization between the signalling and collecting phases. This parallel programming model is more generally referred to as Bulk Synchronous Parallel (BSP) [57].

Algorithm 1 Synchronous execution

```
for i ← 1..num_iterations do
    for all v ∈ V parallel do
        v.doSignal()
    end for
    for all v ∈ V parallel do
        v.doCollect()
    end for
end for
```

This specification allows the efficient execution of algorithms, where every vertex is equally involved in all steps of the computation. However, in many algorithms only a subset of the vertices is involved in each part of the computation. In the next subsection we introduce scoring in order to be able to define a computational model that enables us to guide the computation and give priority to more "important" operations.

Extension: Score-Guided Execution In order to enable the scoring (or prioritizing) of `doSignal()` and `doCollect()` operations, we need to extend the core structures of the SIGNAL/COLLECT programming model and define two additional methods on all vertices $v \in V$:

v.scoreSignal() : Double
is a method that calculates a number that reflects how important it is for this vertex to signal. A scheduler can assume that the result of this method only changes when the `v.state` changes. by default it returns 0 if `state == lastSignalState` and 1 otherwise. This captures the intuition that it is desirable to inform the neighbours iff the state has changed since they were informed last. Note that `lastSignalState` is initially uninitialised, which ensures that by default a vertex signals at least once at the start.

v.scoreCollect() : Double
is a method that calculates a number that reflects how important it is for this vertex to collect. Schedulers assume that the result of this method only changes when `uncollectedSignals` changes. By default it returns `uncollectedSignals.size()`. This captures the intuition that the more new information is available, the more important it is to update the state.

The defaults can be changed to methods that capture the algorithm-specific notion of "importance" more accurately, but these methods should not modify the vertex.

Note: We have the scoring functions return doubles instead of just booleans, in order to enable the scheduler to make more informed decisions. Two examples where this can be beneficial: One can implement priority scheduling, where operations with the highest scores are executed first, or the scheduling could depend on some threshold (for example for PageRank), which allows the scheduler to decide at what level of precision a computation is considered converged.

Now that we have extended the basic model with scoring, we specify a score-guided synchronous execution of a SIGNAL/COLLECT computation in Algorithm 2. There are three parameters that influence when the

Algorithm 2 Score-guided synchronous execution

```
done := false
iter := 0
while (iter < max_iter and !done) do
    done := true
    iter := iter +1
    for all v ∈ V parallel do
        if (v.scoreSignal() > s_threshold) then
            done := false
            v.doSignal()
        end if
    end for
    for all v ∈ V parallel do
        if (v.scoreCollect() > c_threshold) then
            done := false
            v.doCollect()
        end if
    end for
end while
```

algorithm stops: `s_threshold` and `c_threshold`, which set a minimum level of "importance" for the `doSignal()` and `doCollect()` operations to get scheduled and `max_iter`, which limits the number of iterations. The algorithm stops when either the maximum number of iterations is reached or all scores are below their thresholds. In the second case we say that the algorithm has converged. Note that the thresholds are used to configure an algorithm to skip signal/collect operations and setting them too high can lead to imprecise or wrong results.

Extension: Asynchronous Execution We referred to the first scheduling algorithm as synchronous because it guarantees that all vertices are in the same "loop" at the same time. With a synchronous scheduler it can never happen that one vertex executes a signal operation while another vertex is executing a collect operation, because the switch from one phase to the other is globally synchronised.

Asynchronous scheduling removes this constraint: Every vertex can be scheduled out of order and no global ordering is enforced. This means that a scheduler can, for example, propagate information faster by signalling right after collecting. It also simplifies the implementation of the scheduler in a distributed setting, as there is no need for global synchronisation.

Algorithm 3 Score-guided asynchronous execution

```
ops := 0
while (ops < max_ops and ∃v ∈ V(
 v.scoreSignal() > s_threshold or v.scoreCollect() > c_threshold)) do
   S := choose subset of V
   for all v ∈ S parallel do
      Randomly call either v.doSignal() or v.doCollect() iff respective threshold is
      reached; increment ops if an operation was executed.
   end for
end while
```

Algorithm 3 shows a score-guided asynchronous execution. Again, three parameters influence when the asynchronous algorithm stops: `s_threshold` and `c_threshold` have the same function as in the synchronous case; `max_ops`, in contrast, limits the number of operations executed instead of the number of iterations. This guarantees that an asynchronous execution either stops because the maximum number

of operations is exceeded or because it converged. The purpose of Algorithm 3 is not to be executed directly, but to specify the constraints that an implementation of the model has to satisfy during an asynchronous execution. This freedom of allowing an arbitrary execution order is useful, because if an algorithm no longer has to maintain the execution order of operations, then one is able to use different scheduling strategies for those operations.

We refer to the scheduler that we most often use as the "eager" asynchronous scheduler: In order to speed up information propagation this scheduler calls `doSignal` on a vertex immediately after `doCollect`, if the signal score is larger than the threshold.

Distinction: Data-Graph vs. Data-Flow Vertex When using the synchronous scheduler without scoring (Algorithm 1), then the collect function processes the signals that were sent along all edges during the last signalling step. When we introduce scoring, not all edges might signal during every step. There is a similar issue with asynchronous scheduling: While no signal might be forwarded along some edges, other edges might have forwarded multiple signals. For this reason we distinguish two categories of vertices that differ in the way they collect signals:

Data-Graph Vertex A data-graph vertex is most similar to the behaviour of a vertex in the basic execution mode: It processes `v.signals`, all the values in the signal map. This means that only the most recent signal along each edge is collected. If multiple signals were received since signals were last collected, then all but the most recent one are never collected. If no signal was sent along an edge, but there was a previous signal along that edge, then this older signal is collected for that edge. This vertex type is suitable for most graph algorithms.

Data-Flow Vertex A data-flow vertex is more similar to an actor. It collects all signals in `v.uncollectedSignals`, which means that it collects all signals that were received since the last collect operation. This vertex type is suitable for asynchronous data-flow processing and some graph algorithms, such as Delta-PageRank (see 5.1).

3.3 Extension: Graph Modifications and Incremental Recomputation

SIGNAL/COLLECT supports graph modifications during a computation. They can be triggered externally or from inside the `doSignal()` and `doCollect()` methods. This means that vertices and edges can dynamically modify the very graph they are a part of. Modifications include adding/removing/modifying vertices/edges or sending signals from outside the graph along virtual edges (i.e., sending a message to a vertex with a known id without adding an explicit edge to the compute graph).

When an edge is added or removed, a scheduler has to update `scoreSignal()` and `scoreCollect()` of the respective vertex in order to check if the modification should trigger a recomputation. This is enough to support incremental recomputation for many algorithms and modifications. For some algorithms, however, additional recomputations are also required for vertices when *incoming* edges change. The more powerful incremental recomputation scheme described in [5] could be adapted to cover these cases, but would require an additional extension to track changes of incoming edges.

Modifications are applied in per-source FIFO order, which means that all the modifications that are triggered by the same source are applied in the same order in which they were triggered. There are no guarantees regarding the global ordering of modifications.

4 The Signal/Collect Framework — An Implementation

The SIGNAL/COLLECT framework provides a parallel and distributed execution platform for algorithms specified according to the SIGNAL/COLLECT programming model. In this section we explain some interesting aspects of our implementation.

The framework is implemented in Scala, a language that supports both object-oriented and functional programming features and runs on the Java Virtual Machine. We released the framework under the permissive Apache License 2.0[2] and develop the source code publicly, inviting external contributions.

[2] `https://github.com/uzh/signal-collect` and `http://www.signalcollect.com`

4.1 Architecture

The framework can both parallelise computations on multiple processor cores, as well as distribute computations over a cluster. Internally, the system uses the Akka[3] distributed actor framework for message passing.

The different system components such as the coordinator and workers are implemented as actors. The coordinator bootstraps the workers and takes care of global concerns such as convergence detection and preventing messaging overload. Each worker is responsible for storing a partition of the vertices.

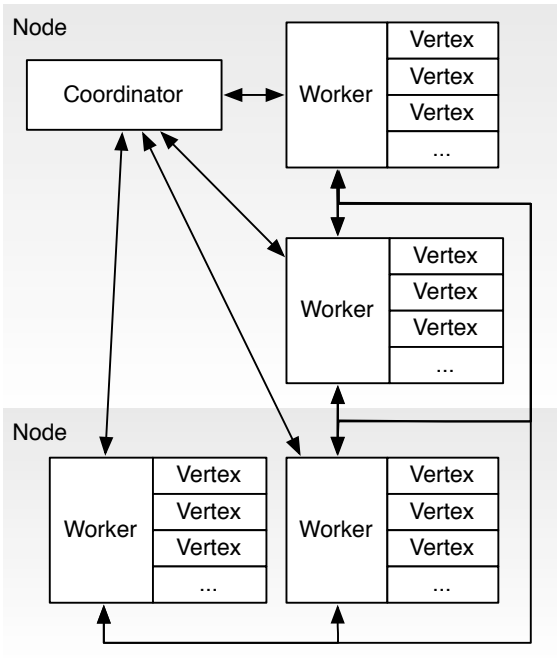


Fig. 2. Coordinator and worker actors, edges represent communication paths. Workers store the vertices.

The scheduling of operations and message passing is done within workers. Figure 2 shows that each node hosts a number of workers and each worker is responsible for a partition of the vertices. Workers communicate directly with each other and with the coordinator. Workers have

[3] `http://akka.io/`

a pluggable scheduler that handles the delivery of signals to vertices and the ordering of signal/collect operations.

Vertices are retrieved from and stored in a pluggable storage module, by default implemented by an in-memory hash map, especially optimised for storing vertices and for efficiently supporting the operations required by the workers.

A vertex stores its outgoing edges, but neither the vertex nor its outgoing edges have access to the target vertices of the edges. In order to efficiently support parallel and distributed execution, modifications to target vertices from the model are translated to messages that are passed via a message bus.

Every worker and the coordinator have a pluggable message bus that takes care of sending signals and translating graph modifications to messages. SIGNAL/COLLECT also has implementations that support features such as bulk-messaging or signal combiners.

4.2 Aggregation Operations

The framework also supports MapReduce-style aggregations over all vertices: The map function is applied to all vertices in the graph. The reduce function aggregates the mapped values in arbitrary order. Aggregation operations are used to compute global results or to define termination conditions over the entire graph.

4.3 Graph Partitioning and Loading

Workers have ids from 0 ascending and by default the graph is partitioned by using a hash function on the vertex ids. This is similar to how graphs are partitioned in most other graph processing frameworks. For large graphs it usually leads to similar numbers of vertices per partition, but also to a large number of edges between partitions. To improve on this we could adopt some of the optimisations used in the Graph Processing System (GPS) [50]. Because computing a balanced graph partitioning with minimal capacity between partitions is a hard problem itself [1], this would mainly improve performance in cases where algorithms are run on the same graph repeatedly, for long-running algorithms, or when messaging bandwidth is the main bottleneck.

The default storage implementation keeps the vertices in memory for fast read and write access. Extensions for secondary storage can be implemented [53]. Graph loading can be done sequentially from a coordinator actor or preferably in parallel, where multiple workers load parts of the graph at the same time. Specific partitions can be assigned to be loaded by particular workers. This can be used to have each worker load its own partition, which increases the locality of the loading.

4.4 Flexible Tradeoffs

Our framework has defaults that work for a broad range of algorithms, but are not the most efficient solution for most of them. These default implementations can be replaced allowing a graph algorithm developer to choose the trade-off between implementation effort and resulting performance.

An example of a tradeoff is the propagation latency vs. messaging overhead: While sending each signal as soon as possible leads to a low latency and can perform well in local computations, sending each signal by itself will cause a lot of overhead in a distributed setting. Our implementation allows to plug in a custom bulk scheduler and bulk message bus implementation to choose a trade-off that suits the use case (throughput vs. latency).

In spite of this flexibility, our SIGNAL/COLLECT implementation is optimised for sparse graphs, due to the internally used adjacency list structure. For very dense graphs alternative representations, for example as a compressed matrix, might perform better.

4.5 Convergence and Termination

The framework has to decide when an algorithm execution ends. It is not in general possible to say which algorithms can converge: The SIGNAL/COLLECT functions allow for arbitrary code execution and is hence subject to the halting problem.

For this reason the question of convergence and termination are algorithm-specific. The framework terminates as soon as an algorithm has converged, according to the score-guided execution definitions in Section 3. The framework also allows for other termination conditions in case convergence was not reached before another condition: One can give a

step limit for synchronous computations and a time limit for both synchronous and asynchronous computations. The framework also supports convergence criteria based on global aggregations that are executed in step intervals for synchronous computations or in terms of time intervals for asynchronous computations.

There is also a continuous asynchronous mode where the framework keeps running and executes operations incrementally as they are triggered by modifications and signals. This mode is used by TripleRush [55] and raises the question of how to detect when a query has finished executing, given that the execution can branch many times and the number of signals/results is usually not known a priori. We solved this by implementing per-query convergence detection on top of SIGNAL/COLLECT: Every query carries a number of tickets with it and when the execution branches the tickets are split up among all branches. The vertex that does the final result reporting knows how many tickets to expect in total and can report success when all the initially sent out tickets have arrived. We described this case in more detail, because it displays that convergence detection is algorithm-specific and that there is a lot of flexibility when it comes to determining convergence criteria: it is even possible to build a custom convergence detection on top of the SIGNAL/COLLECT model.

5 Evaluation

In this section we evaluate the programming model, the scalability of our implementation, and the impact of asynchronous scheduling. The different contributions require different research methods: We evaluate the programming model by adapting important algorithms in a few lines of code. In addition to the expressiveness, we also show that our implementation is able to transparently scale algorithms by empirically measuring the speedup when running algorithms while varying the number of worker threads and cluster nodes. Finally, we compare the impact of asynchronous scheduling versus synchronous scheduling on different graphs and algorithms.

5.1 Programming Model

One of our main contributions is the simple, compact, yet expressive programming model. Whilst simplicity of a program is difficult to judge objectively, compactness and expressiveness are easier to show.

We demonstrate the expressiveness by giving adaptations of ten algorithms from categories as varied as clustering, ranking, classification, constraint optimisation and query processing.

Most algorithms are presented in a simplified version, more advanced versions of many of the examples are available online.[4] To enhance readability and facilitate the comparison of different algorithms, each algorithm is structured in a table representing the three core elements of a computation: The *initial state* represents the state of the vertices when they get added to the graph. The *collect* method uses the vertex state, the appropriate signals for the vertex type, and other vertex attributes/methods to compute a new vertex state. The *signal* method uses attributes/methods defined on the source vertex and edge attributes/methods to compute the signal that is sent along the edge. All described algorithms work on homogeneous graphs that use only one type of vertex/edge, which is specified in the table. Additional information and explanations for complex functions are provided in the algorithm descriptions.

Unless stated otherwise, the described algorithms use the default `scoreSignal()` and `scoreCollect()` implementations for automated convergence detection.

PageRank This graph algorithm computes the importance of a vertex based on the link structure of a graph [46]. The vertex state represents the current rank of a vertex (Figure 3). The signals represent the rank transferred from the source vertex to the target vertex. The vertex state is initialised with the `baseRank = 0.15` and the damping factor is usually set to `0.85`.

Convergence: If one looks how the initial rank from a single vertex spreads, then one notices that it decays with the damping factor on every hop and that it eventually tapers off to zero. The computation on all vertices can be seen as many such single-source PageRank computations (sometimes referred to as personalised PageRank) that are overlaid and will, hence, also converge. In practice the convergence to zero can take many iterations, especially when there are cycles present. For this reason we usually set the scoreSignal function to return the delta between the current state of the vertex and the last signalled state (often referred to as the residual). This allows to conveniently set the desired level of preci-

[4] `https://github.com/uzh/signal-collect/tree/master/src/test/scala/com/signalcollect/examples`

sion by for example setting a signal threshold of 0.001. This means that a vertex will only signal if the residual is still larger or equal to 0.001.

`initialState`	`baseRank`
`collect()`	`return baseRank + dampingFactor * sum(signals)`
`signal()`	`return source.state * edge.weight / sum(edgeWeights(source))`

Fig. 3. PageRank (data-graph)

It is also possible to implement PageRank as a *data-flow algorithm* by signaling only the rank deltas (Figure 4). This version can be further

`initialState`	`baseRank`
`collect()`	`return oldState + dampingFactor * sum(uncollectedSignals)`
`signal()`	`stateDelta = source.state - source.signaledState` `return stateDelta * edge.weight / sum(edgeWeights(source))`

Fig. 4. Delta PageRank (data-flow).

optimised by not sending the source vertex id with the signals, which saves bandwidth.

Single-source shortest path (SSSP) This algorithm computes the shortest distance from one special source vertex (`S`) to all the other vertices in the graph. The vertex states represent the shortest currently known path from `S` to the respective vertex (Figure 5). Edge weights are used to represent distance. The signals represent the total path length of the shortest currently known path from `S` to `edge.target` that passes through `edge`.

Convergence: Vertices only signal if their distance was lowered by an incoming signal. Given that the distances can never be smaller than zero and that the first change of a vertex's state will set it to a finite number,

the distance of a vertex can only be lowered a finite number of times, which means that the algorithm is guaranteed to eventually converge.

`initialState`	`if (isSource) 0 else infinity`
`collect()`	`return min(oldState, min(signals))`
`signal()`	`return source.state + edge.weight`

Fig. 5. Single-source shortest path (data-graph/data-flow).

Vertex Colouring A vertex colouring problem is a special constraint optimisation problem that is solved when each vertex has an assigned colour from a set of colours and no adjacent vertices have the same colour. The following simple and inefficient algorithm solves the vertex colouring problem by initially assigning to each vertex a random colour from some arbitrary set of colours (Figure 6). Then, the vertices check if their own colour (state) is already occupied (contained in the collection of received signals). If such a conflict is encountered, they switch to a random colour except their current colour. The default `scoreSignal()` method ensures that the vertex signals again if there was a conflict. If there was no conflict, then the vertex stays with its current colour.

Algorithms such as this one can solve many optimisation problems such as scheduling or finding solutions for Sudoku puzzles. The described algorithm works with undirected edges. In SIGNAL/COLLECT these are modelled using two directed edges. The performance could be improved by using a better optimisation algorithm of which many have SIGNAL/COLLECT adaptations.[5]

Convergence: The computation keeps on going until there are no more conflicts between colours. If the number of colours available is smaller than the chromatic number of the graph, then there is no solution without conflicts, which means that this algorithm is not guaranteed to converge.

[5] A Distributed Stochastic Algorithm implementation in SIGNAL/COLLECT, for example, can be found at: `https://github.com/elaverman/signal-collect-dcops/blob/2e25766c04d66a6cdce4ec5a659fb0dfc45436d6/src/main/scala/com/signalcollect/approx/flood/DSA.scala`

`initialState`	`randomColour`

`collect()`	`if (contains(signals, oldState))`
	`  return randomColorExcept(oldState)`
	`else`
	`  return oldState`

`signal()`	`return source.state`

Fig. 6. Vertex colouring (data-graph).

Label Propagation This iterative graph clustering algorithm assigns to each vertex the label that is most common in its neighbourhood [64]. Our variant is called Chinese Whispers Clustering [3] and has applications in natural language processing. The algorithm works on graphs with undirected edges which are modelled with two directed edges.

The vertex state represents the current vertex label (= cluster) and it is initialised with the vertex id (Figure 7). This means that each vertex starts in its own cluster. Then, labels are propagated to neighbours. When a vertex receives neighbours' labels, it appends its own label to the collection of labels signalled by the neighbours. It then updates its own label to the most frequent label in that extended collection. Ties are broken arbitrarily.

The convergence depends on the mostFrequentValue function: According to [3] the algorithm does not converge if that function does not break ties in a consistent way, but that only a few iterations are needed until almost-convergence.

`initialState`	`id`
`collect()`	`return mostFrequentValue(append(oldState, signals))`
`signal()`	`return source.state`

Fig. 7. Label propagation (data-graph).

Relational Classifier Relational classification can be considered a generalisation of label propagation. The presented classifier (Figure 8) is a variation of the probabilistic relational-neighbour classifier [39, 40]. The

algorithm works on graphs with undirected edges which are modelled with two directed edges. `ProbDist` represents a probability distribution over different classifications. Each vertex starts with an initial probability distribution over the classes, which can be uniform or can reflect the observed frequencies of classes in the training set. If the class of a vertex is available as training data, then that class gets probability 1 and is not changed by the algorithm. When a vertex receives the distributions of its neighbours, then it updates its own distribution to a normalised sum of the class probability distributions of the neighbours. A collective inference scheduling can be chosen using the `scoreSignal()` implementation to determine when a vertex informs its neighbours about its label distribution. This classifier only works when edges are more likely to connect vertices that belong to the same class (homophily) and, given its supervised nature, when some classes are known as training data [39]. The algorithm described here can be extended with a local classifier to determine the initial state. This initial state would be added to the sum of neighbour states in the `collect` method (potentially with a higher weight). In this case, it is no longer necessary for some classes to be known. The homophily constraint could be dropped by using a more advanced relational classification algorithm [6, 15].

Convergence: According to [39] there is no guarantee of convergence, but according to [40] one can extend the algorithm with simulated annealing to ensure and control convergence.

`initialState`	`if (isTrainingData) knownValue else avgProbDist`

`collect()`	`if (isTrainingData)`
	`  return oldState`
	`else`
	`  return signals.sum.normalise`

`signal()`	`return source.state`

Fig. 8. Relational classifier (data-graph).

Conway's Game of Life (Life) Life is played on a large checkerboard of cells, where each cell can be in one of two states (dead/alive) [14]. The game progresses in turns and each turn the state of a cell is updated based on the states of its neighbouring cells. The game is mapped to SIGNAL/-

COLLECT by representing each cell as a vertex, alive is represented with state 1 and dead as state 0 (Figure 9). The neighbourhood relationships between the cells are modelled with edges between neighbouring cells. A vertex counts how many of its neighbours are alive and uses this to determine its state next turn according to the rules of the game. Life is famous for the complex patterns that can emerge from its simple rules.

Convergence depends on the initial configuration and there are many (famous) initial configurations that will never converge.

`initialState`	`if (isInitiallyAlive) 1 else 0`

`collect()`	`switch (sum(signals))`
	`  case 0: return 0  // dies (too lonely)`
	`  case 1: return 0  // dies (too lonely)`
	`  case 2: return oldState // same as before`
	`  case 3: return 1  // becomes alive`
	`  other:  return 0  // dies (too crowded)`

`signal()`	`return source.state`

Fig. 9. Conway's Game of Life (data-graph).

Threshold Models of Collective Behaviour Granovetter [17] describes threshold models of collective behaviour to model situations in which agents have two options and the risk/payoff of each option depends on the behaviour of neighbouring agents. The risk/payoff is determined by a threshold which can be different for every actor. Threshold models allow to model the collective behaviour of a group of actors. Granovetter uses the example of rioting, but argues that such models can also be used to model innovation, rumour diffusion, disease spreading, strikes, voting, migration, educational attainment, and attendance of social events.

Such models can be mapped to SIGNAL/COLLECT by representing each agent with a vertex and connecting all agents that can observe each other's behaviour with edges. The initial state determines the default behaviour of an agent (Figure 10). In our example, an actor does not riot initially, unless it is a natural rioter, which means that the agent would riot no matter what its neighbours do. Edges inform neighbours about the behaviour of an agent and in the `collect()` method the agent analyses what fraction of its neighbours are taking the alternative decision.

If the fraction of neighbouring agents that display a behaviour exceeds the individual threshold (in our example `riotingThreshold`), then the agent changes its behaviour and switches to the alternative behaviour (e.g., start to riot).

Convergence: In the described model no person will ever stop rioting, once they decide to riot. For this reason only a finite number of 'I am now rioting' messages can ever be sent. This means that at some point either everyone is rioting, or the non-rioters will never receive an additional message that could shift them towards becoming rioters. For this reason, the computation is guaranteed to converge.

`initialState`	`if (isNaturalRioter) true else false`

`collect()`	`riotingNeighbours = filterTrue(signals)`
	`rioterFraction = riotingNeighbours.size / signals.size`
	`if (rioterFraction > riotingThreshold)`
	`  return true`
	`else`
	`  return false`

`signal()`	`return source.state`

Fig. 10. Threshold models of collective behaviour (data-graph).

Matching Path Queries This algorithm matches path queries, which is a typical use case for a graph processing system. The signals sent in the algorithm initially come from outside the graph along a virtual edge. The signals are path queries that specify a pattern of vertices and edges that they can match. An example for such a pattern might be: Match any path that starts with a vertex that has a "professor" property, continues along an edge that has an "advises" property and ends with an arbitrary vertex.

Once a query arrives at a vertex, its first part is matched with the vertex at which it has arrived (Figure 11). This is done with the `successfulMatchesWithVertex()` function, which returns only the queries that have successfully matched with the local vertex.

If the query is fully matched—meaning all parts of its path are bound to a vertex or edge—then this path is reported as a result (this could be done by adding it to some result attribute that is later picked up by an

aggregation operation). If there is still a part of the query left that needs to be matched, then it is added to the state set of the vertex. During the signal operation all edges try to match the next part of the queries—the one potentially constraining the type of edge to follow—using their `successfulMatchesWithEdge()` functions. Queries that were successfully edge-matched are returned by that function and signalled along the respective edges.

Matching path queries has many use cases: From simpler ones such as triangle/cycle detection, the approach could be extended to more complex tasks such as computing random walks with restarts or even matching expressive graph query languages.

Convergence: If each query only has a finite number of expected vertex/edge matches, then the execution is guaranteed to converge, because all queries will eventually either be eliminated or become fully matched.

`initialState`	`emptySet`

`collect()`	`matched = successfulMatchesWithVertex(signals)`
	`(fullyMatched, partiallyMatched) =`
	`partition(matched)`
	`reportResults(fullyMatched)`
	`return union(oldState - lastSignalState,`
	`partiallyMatched)`

`signal()`	`return successfulMatchesWithEdge(source.state)`

Fig. 11. Matching path queries (data-flow).

Artificial Neural Networks Artificial neural networks are the result of an attempt to imitate the structure of biological neural networks and there are *"literally tens of thousands of published applications"* [49, p. 748]. Neural networks consist of nodes connected by links [49, p. 737]. The nodes are mapped to vertices in SIGNAL/COLLECT and the links are mapped to directed edges. Activations are sent as signals between edges, with the difference that they already get adjusted for edge weight in the `signal()` method of the edge (Figure 12). The activation function is mapped to the `collect()` method and updates the state that represents the unit activation. Varying inputs are sent from outside the graph as signals along virtual edges.

Convergence: Neural networks usually do not contain cycles, so if there are no more new inputs, then the remaining activations are guaranteed to finish propagating through the network at some point.

`initialState`	`0`
`collect()`	`return 1 / (1 + e^-signals.sum)`
`signal()`	`return source.state * edge.weight`

Fig. 12. Artificial neural networks (data-graph).

Sketching of some additional algorithms The "Bipartite Matching" and "Semi-Clustering" algorithms described in the Pregel paper [41] can be adapted to the SIGNAL/COLLECT model by separating the compute function into a signal part for communication and a collect part for the state update. They require access to the step number, which is not available in the default SIGNAL/COLLECT model, but can be added for example by using parallel update operations between computation steps. An adaptation of "Loopy Belief Propagation" has been outlined in [54] and was used to do inference on Markov logic networks [51].

We have also implemented a triple store with competitive performance inside SIGNAL/COLLECT [55]. This system uses three different vertex types to model an index and to keep track of query executions.

The main benefit of adapting an algorithm or system to the SIGNAL/COLLECT model is that its execution is automatically scheduled in parallel (or even distributed, if several machines are available), which allows for scalability. In the next section we empirically evaluate the scalability of our framework when executing algorithms expressed in the programming model.

5.2 Scalability

In this subsection we evaluate the scalability of the SIGNAL/COLLECT framework. In order to evaluate this we empirically measure the performance of our framework on multiple algorithms while varying the available resources. More specifically, we analyse the scalability by varying (1) the number of worker threads, (2) the size of the processed graph, and (3) the number of cluster nodes.

Multi-core (vertically/scale up) We determined the multi-core scalability by measuring the parallel speedup when running an algorithm on the same graph, but with a varying number of worker threads.

In a first benchmark we ran the SSSP and PageRank algorithms on a machine with two twelve-core AMD Opteron™ 6174 processors and 66 GB RAM. We executed these algorithms with both synchronous and "eager" asynchronous scheduling. They were run on the web graph dataset[6] with 875 713 vertices (websites) and 5 105 039 edges (hyperlinks). Each combination of algorithm and scheduler was run whilst varying between 1 and 24 worker threads (as the machine has 24 cores). Each run was executed ten times and we graphed the resulting average running time in Figures 13 (PageRank) and 14 (SSSP), where the error bars indicate min/max running times. The speedup was calculated relative to the average runtime with one worker thread. Each execution was run cold in a new JVM, because this reflects the actual usage best.

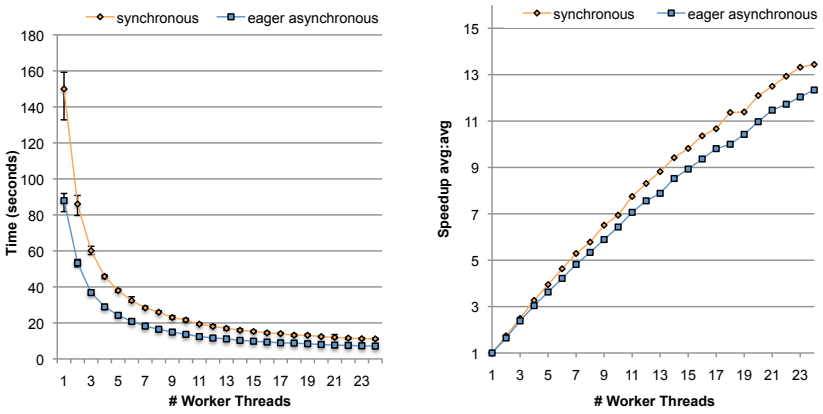
PageRank was run with a signal function that returns the delta between the previous signal state and the current state (see Figure 4). The signal threshold was set to 0.01, which determines the precision of the result. More detailed evaluation parameters can be found in the evaluation program.[7]

The fastest running time was 7 seconds for PageRank and 1.2 seconds for SSSP. The speedup when going from 1 core to 24 cores was 9 for SSSP and around 13 for PageRank. This shows that SIGNAL/COLLECT scales with additional cores, but that the actual factor depends on the algorithm. The achievable speedup also depends on the graph structure: Running SSSP on a chain of vertices would not allow for any parallelism with our implementation.

Data scalability In order to evaluate how SIGNAL/COLLECT scales with increasing graph sizes we had to determine what kind of graphs to evaluate it on. Given that we want the graphs to be as similar as possible, Kronecker graphs [34] seem like a good choice, because they preserve

[6] `http://snap.stanford.edu/data/web-Google.html`

[7] The evaluation program used was MulticoreScalabilityEvaluation in `https://github.com/uzh/signal-collect-evaluation` at revision 05057d000d. The snapshot dependencies were `https://github.com/uzh/signal-collect` at revision ba26e95e20 and `https://github.com/uzh/signal-collect-graphs` at revision 0149927e68. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldEVwYk1lWDJpQmVRc0QtUWxLcFVXUWc`.



(a) Execution Times (b) Speedup

Fig. 13. Multi-Core Scalability of PageRank

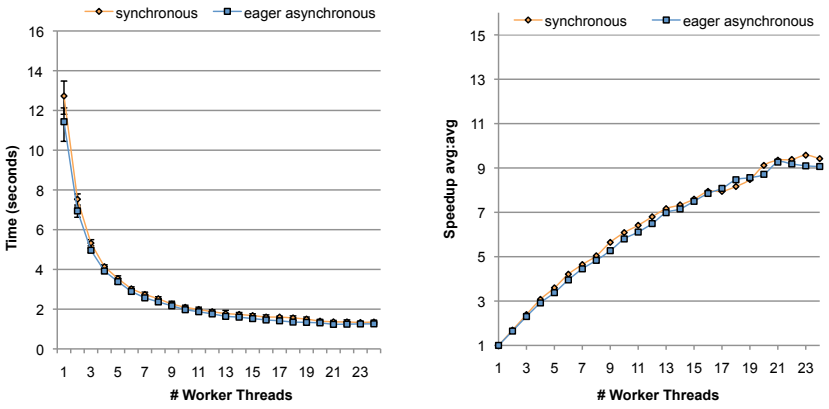
many properties of a graph across different scales. We used the reference implementation of the generator that is part of SNAP [2].[8]

We generate the graphs by using the fitted parameters for the Notre Dame web graph ([0.999 0.414; 0.453 0.229]) which we also got from [34]. With these parameters we generated graphs with between 20 to 26 iterations of the Kronecker product, which resulted in graphs with between 659 518 vertices and 2 652 653 edges up to 39 865 268 vertices and 224 276 985 edges, in between increasing approximately with powers of two. For 27 iterations the graph generator repeatedly threw errors (the machine on which it was run had 128 GB of RAM).

The partitions of the generated graphs are in many ways unbalanced: The number of outgoing edges from a worker partition varies by a factor of around 9 between the worker with the most outgoing edges and the one with the fewest. With regard to message sending, the distribution is even more uneven: We checked the number of sent messages between the busiest worker for one run at 20 and 26 iterations, and in both cases the busiest worker sent more than 100 times as many messages as the least busy worker.

We ran the experiment on machines with 128 GB RAM and two E5-2680 v2 processors at 2.80GHz, with 10 cores per processor. The JVM

[8] `https://snap.stanford.edu/snap/`



(a) Execution Times (b) Speedup

Fig. 14. Multi-Core Scalability of Single-Source Shortest Path

on the machines used between 300 MB (minimum on the smallest graph) and 12.5 GB of RAM (maximum on the largest graph).

Figure 15 shows the performance when running Delta PageRank (subsection 5.1) with a threshold of 0.01 on these synthetic graphs.[9] We ran all evaluations 10 times and plot the average execution time. We tested the running times with both a specialised signal combiner for the rank deltas and with the generally applicable bulk messaging. The plot is logarithmic in both axes and it shows that SIGNAL/COLLECT scales linearly with increasing dataset sizes. Whilst bulk messaging performs better for smaller graph sizes, the signal combiner is faster on the largest dataset.

Distribution (horizontally/scale out) We determined the distributed scalability by measuring the speedup when running an algorithm on the same graph, but with a varying number of cluster nodes.

For this benchmark we ran Delta PageRank on the Yahoo! AltaVista webgraph[10] with 1 413 511 390 vertices and 6 636 600 779 edges. This is

[9] The evaluation program used was PageRankEvaluation in `https://github.com/uzh/signal-collect-evaluation` at revision de1a018. The used `https://github.com/uzh/signal-collect` dependency was at revision 77da25f. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldFFxaFp1N0RzTDBKdzhoSHlmb3M1T0E`.

[10] Yahoo! Academic Relations, Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`

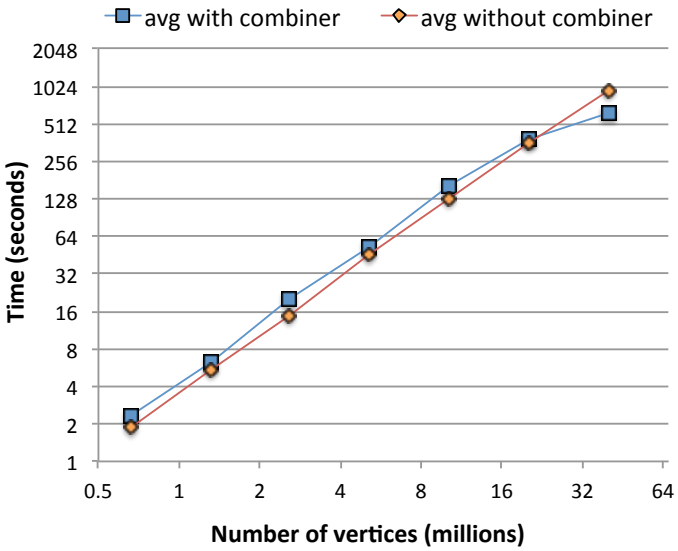


Fig. 15. Data scalability for PageRank on synthetic Kronecker graphs. Both axes have logarithmic scales.

one of the largest real-world graphs available for such evaluations and a realistic use case for the PageRank algorithm. In order to measure scalability we ran the algorithm ten times with each configuration on a cluster with 4, 6, 8, 10, and 12 nodes, with 24 worker threads per node (i.e., from 96 workers up to 288 workers). The nodes were the same 24-core machines used in the multi-core scalability evaluation connected by a 1 gigabit ethernet switch. In all computations the coordinator actor was running on a laptop on a different network, this machine had no problem handling the load of running termination detection and flow control. The latency between coordinator and workers was less than one millisecond. Given that the heartbeat interval was 100 milliseconds, it is unlikely for the remoteness of the coordinator to have had much effect on the computation.

The vertices were partitioned using hashing as described earlier, so most edges spanned different workers and nodes. The graph was loaded from the local file system of the machines and loading took between 45 seconds (fastest run with 12 nodes) and 235 seconds (slowest run with 4 nodes).

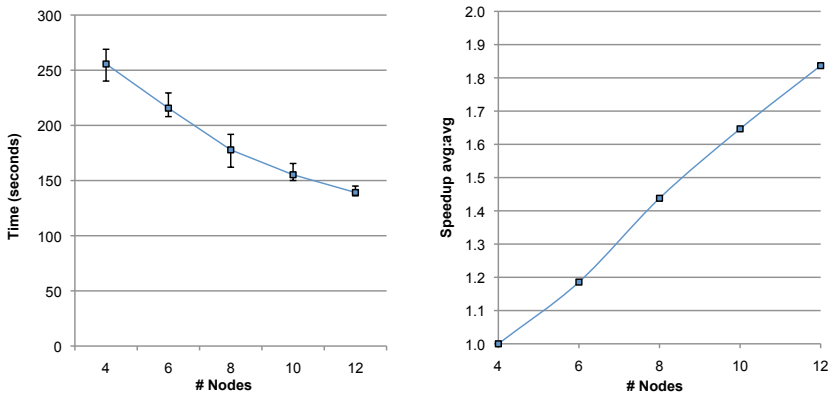
The huge number of signals required more efficient usage of bandwidth, which is why we used a bulk scheduler and bulk message bus. When scaling across more nodes and workers, this means that either each bulk signal has to contain fewer signals or that there is increased latency that would impair algorithm convergence. We chose to keep the latency constant, which has the effect of reducing the benefits of bulk signalling for runs with more nodes, but removes convergence characteristics as a confounding factor.

Another optimisation that we used was to have the vertex change the edge representation and for each edge only store the id of the target vertex. The target ids are integers, so to further reduce the memory footprint we sorted the array of target ids and at each position only stored the delta from the previous array entry. We then took advantage of the smaller ids by using variable length encoding on the id deltas. Furthermore, we collected signals right when they were delivered, which makes it unnecessary to store them inside the vertex until they are collected. These optimisations reduced the memory footprint and allowed us to successfully run the algorithm on only four machines.

Finally, Delta PageRank uses a signal function that returns the delta between the previous signal state and the current state. Every execution was run until convergence with a signal threshold of 0.01 and both the vertex state (PageRank) as well as the signals were represented as floating point numbers.

Figure 16(a) graphs the execution times. It shows that increasing the number of nodes decreases run-time. Indeed, the speedup plot in Figure 16(b) shows that for using three times more resources we get a speedup of almost two. This is decent, considering that more nodes means a larger fraction of signals are sent across nodes (over the slow network, as opposed to fast in-memory transfers) and that there is more overhead for signalling due to smaller bulk signal sizes. More detailed evaluation parameters can be found in the evaluation program.[11]

[11] The evaluation program used was DistributedWebGraphScalabilityEval in `https://github.com/uzh/signal-collect-evaluation` at revision 701e208. The snapshot dependencies were `https://github.com/uzh/signal-collect` at revision 43c3b0ffe7 and `https://github.com/uzh/signal-collect-graphs` at revision f35637c930. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldDF2dEJIWnliVkJ0cjBrVlVvOTBkMkE`.



(a) Execution Times (b) Speedup

Fig. 16. Horizontal scalability of PageRank on the Yahoo! AltaVista Web Page Hyperlink Connectivity Graph with 1 413 511 390 vertices and 6 636 600 779 edges. The data points in 16(a) show the average execution time over 10 runs and the error bars indicate the fastest and slowest runs. 16(b) plots the speedup relative to the average execution time with 4 nodes. The signal threshold used was 0.01, state and signals were represented as floats.

Comparison with other reported results We are well aware that comparing run-times between systems run on different machines is a problematic proposition at best. The main goal of the comparisons below is, therefore, to provide an intuition of the order of scalability and performance of SIGNAL/COLLECT in contrast to other systems reported on in the literature.

Pegasus [29] is a MapReduce-based system that ran 10 iterations of an iterative belief propagation algorithm on the same Yahoo! AltaVista webgraph using 100 machines of a Hadoop cluster. This computation took 4 hours.

GPS [50] computed 50 iterations of PageRank on a webgraph with 51 million vertices and 1.9 billion edges in *846 seconds* using a cluster of 60 Amazon EC2 nodes (4 virtual cores and 7.5GB of RAM each). Using a pre-partitioned graph reduced the computation time to 372 seconds. In their evaluations they describe that GPS runs more than an order of magnitude faster than Giraph.

GraphLab did not report any evaluations for the PageRank algorithm, which complicates comparison. The largest (pre-partitioned) graph it was

evaluated on had 27 million vertices and 375 million edges. The non-partitioned ones were smaller.[12]

PowerGraph [16] required about 14 seconds to compute PageRank on a Twitter follower graph with *40 million vertices* and 1.5 billion edges employing a cluster of 64 Amazon EC2 nodes (8 cores and 23 GB of RAM each, connected by 10 gigabit Ethernet). They report faster times with coordinated partitioning requiring an up-front loading time of more than 200 seconds. In their evaluations PowerGraph is at least an order of magnitude faster than the other frameworks they compare against.

5.3 Performance comparison with PowerGraph

In order to fairly compare the performance of two systems they have to be run on the same hardware. To that end we ran PageRank on both PowerGraph and SIGNAL/COLLECT, again on the AltaVista webgraph. The experiment was run on a cluster of 8 machines, each machine having 128 GB RAM and two E5-2680 v2 processors at 2.80GHz, with 10 cores per processor. The machines were connected with both 10Gbps Ethernet and 40Gbps Infiniband. For PowerGraph we used the repository version from July 6th 2014.[13] From that version we used the existing toolkit PageRank implementation.[14] For SIGNAL/COLLECT we used a precise version of the asynchronous Delta PageRank, as in the scalability evaluation, but in addition we used a message combiner that sums up ranks on the sender side.[15] Both systems use doubles internally to represent ranks and for the baseline we configured them with a tolerance of 0.001 and with enabled Infiniband. Baseline PowerGraph was run with the synchronous engine, whilst baseline Delta PageRank was run with the asynchronous execution

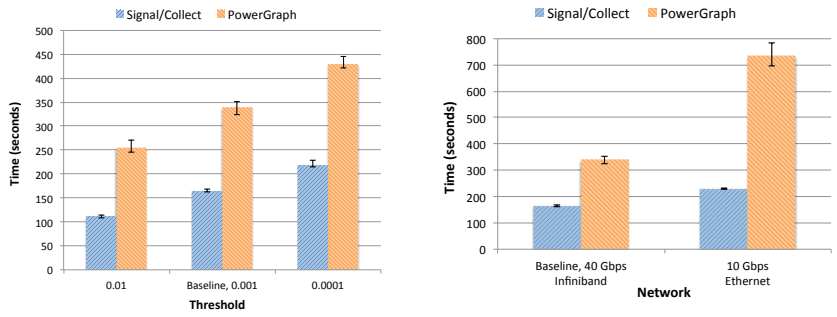
[12] GraphLab did not scale up to the Yahoo! AltaVista webgraph according to the thesis defence slides of Joseph E. Gonzalez, slide 70, `http://www.cs.cmu.edu/~jegonzal/talks/jegonzal_thesis_defense.pptx`.

[13] `https://github.com/graphlab-code/graphlab/tree/4d201c2599d51f9975617dc4a3f39f6c9d489cc4`

[14] `https://github.com/graphlab-code/graphlab/blob/4d201c2599d51f9975617dc4a3f39f6c9d489cc4/toolkits/graph_analytics/pagerank.cpp`

[15] The evaluation program used was PageRankEvaluation in `https://github.com/uzh/signal-collect-evaluation` at revision d3c7aaa. The used `https://github.com/uzh/signal-collect` dependency was at revision 17a28a2. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldFhUWHM2dG13emZkaUNhVGhreWZBNlE`.

mode of SIGNAL/COLLECT. We compared the best-performing configurations for the baseline (with the exception of the threshold parameter).



(a) (b)

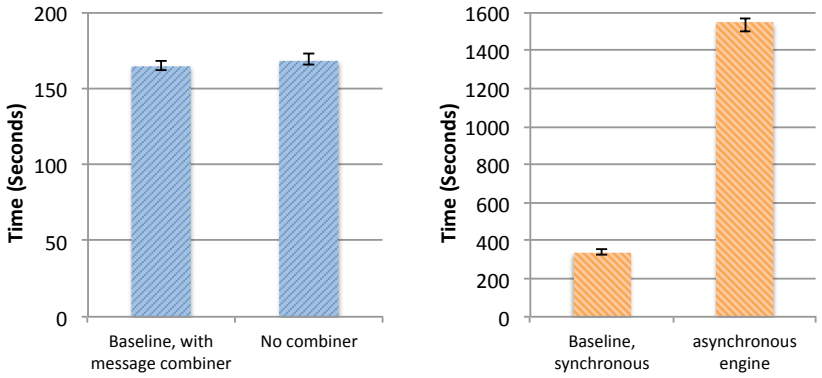
Fig. 17. Comparison between SIGNAL/COLLECT and PowerGraph of the respective execution times for computing PageRank on the Yahoo! AltaVista webgraph when (a) the convergence threshold and (b) the network is varied relative to the baseline configuration.

In Figures 17 and 18 the bar displays the average running time over ten runs and the error bars indicate the performance of the fastest and slowest runs.

Figure 17(a) shows how the execution times change when the convergence threshold is varied around the baseline. We see that SIGNAL/COLLECT takes about half as long at all precision levels.

Figure 17(b) shows the comparison of the baseline configurations of PowerGraph and SIGNAL/COLLECT with one in which Infiniband is disabled. We notice that disabling Infiniband increases the running time of SIGNAL/COLLECT by around 50%, whilst the PowerGraph execution time more than doubles. We were surprised by this, because for this graph PowerGraph reported a replication factor of 3.27, which means that for each original vertex there were on average more than 3 replicas. We expected that this would lead to a lower sensitivity to network bandwidth and latency, but the results suggest that PowerGraph is more sensitive to the quality of the network.

We also tested enabling delta caching for the PowerGraph synchronous and asynchronous engine. In both cases enabling delta caching resulted in the computation still running after several hours and we canceled the evaluation at that point.



(a) (b)

Fig. 18. Figure 18(a) compares the baseline execution time of SIGNAL/COLLECT, which has an enabled message combiner, with the generally applicable bulk messaging. Figure 18(b) compares the baseline execution time of PowerGraph, which uses the synchronous engine, with the asynchronous engine of PowerGraph. Note that Figure 18(a) and Figure 18(b) do not use the same scale on the y-axis.

We loaded the graphs from different formats and from a network drive, which is why the loading times are not comparable and are subject to variance due to local caching of the files. For SIGNAL/COLLECT we loaded the graph from a binary format that split vertices according to their hash code, which is why each worker only loaded local vertices. For this reason most SIGNAL/COLLECT runs loaded the graph in around 50 seconds. PowerGraph loaded the graph with the automatically determined 'grid' ingress method an adjacency list format from 1413 splits in more than 10 minutes, but this could likely be sped up by a lot if one were to also use a specialised format.

Note: PowerGraph only ran the computation on 720 242 173 vertices, because vertices without incoming or outgoing edges were discarded during the loading phase. One SIGNAL/COLLECT run and four PowerGraph runs crashed during the computation, in this case we simply restarted that run.

Memory usage: The PowerGraph allocator reported that it used around 80 GB of the heap on each machine. The memory usage per node of SIGNAL/COLLECT varied between 20 GB and 47 GB, the variance is most likely due to garbage collection timing.

We also tried out alternative configurations: Figure 18(a) compares the impact of using bulk messaging instead of the baseline signal combiner with SIGNAL/COLLECT. We see that bulk messaging is slightly slower, but that the specialised message combiner did not improve the execution time by much. Figure 18(b) compares the execution time when using the PowerGraph asynchronous engine instead of the baseline synchronous engine. We see that the asynchronous engine is more than four times slower. We found this surprising, but a PowerGraph author explains that this observation might be explained by the the asynchronous engine being less optimised.[16]

As always, given the complexity of such distributed frameworks, it is difficult to draw hard conclusions. Nonetheless, we feel that we can clearly state that the SIGNAL/COLLECT approach seems highly competitive when computing a popular algorithm on a web-scale graph using a conventional cluster. We can also state that SIGNAL/COLLECT's retained its competitiveness when turning off some of its more complex refinements (see Figure 18(a)) and that it was less reliant on a fast network for its performance.

5.4 Asynchronous Computation

In this section discuss and evaluate some aspects of asynchronous scheduling in SIGNAL/COLLECT. Depending on the scheduling strategy an asynchronous scheduler can have the following advantages:

- *Lower latency between operations*
  An asynchronous scheduler is not tied to a global ordering that prescribes when information can be propagated. This flexibility can be used to reduce the latency between collecting and signalling and is especially important for use cases such as query processing, where latency is critical (see path query processing 5.1). It can also lead to fewer signal operations due to faster information propagation (see the PageRank scheduler analysis in this subsection).

- *Reduction of oscillations*
  Some synchronous algorithms can be prone to getting trapped in oscillation patterns, where vertices cycle through states in lockstep.

[16] `http://forum.graphlab.com/discussion/comment/277`

Asynchronous processing can reduce such oscillations and allows some of these algorithms to converge quickly).

In order to measure how much impact the lower latency signal propagation has we ran PageRank and SSSP with both kinds of schedulers in the previously described scalability experiments. As reported in Figures 13 and 14 asynchronous scheduling is on average between 36% (with 24 workers) and 41% (with 1 worker) faster than synchronous scheduling. This is largely because of earlier signal propagation: In all cases the asynchronous version required on average 30% fewer signal operations until convergence.

Scheduling does not have the same impact on all algorithms: The single-source shortest path algorithm took approximately the same amount of time, regardless of the scheduling.

To evaluate the *impact of scheduling on oscillations* we ran a greedy algorithm to solve vertex colouring problems on the Latin Square dataset.[17] The graph is a vertex matrix with 100 columns and 100 rows, where all vertices in each column and all vertices in each row are connected (modelled by almost 2 million undirected edges in SIGNAL/COLLECT). The problem requires at least 100 colours to be solved and becomes easier to solve when more colours are available. We ran the algorithm with both synchronous and "eager asynchronous" scheduling for a varying number of available colours. The hardware used was the same as in subsection 5.2. In Figure 19 we show the fastest execution time of the ten runs for each number of colours. Executions were terminated after 20 minutes, even if no solution was found.[18] Each vertex was initialised with the same colour, initialising with random colours would lead to faster executions.

Figure 19 shows the executions with "eager" asynchronous scheduling found solutions much quicker than synchronous executions. For the harder problems with fewer colours there are also several cases where a synchronous scheduling fails to find a solution within the time limit,

[17] We used the dataset provided by CMU at `http://mat.gsia.cmu.edu/COLOR04/INSTANCES/qg.order100.col`

[18] The evaluation program used was VertexColoringSyncVsAsyncEvaluation in `https://github.com/uzh/signal-collect-evaluation` at revision f53d9897b1. The snapshot dependencies were `https://github.com/uzh/signal-collect` at revision 40d89ba1c1 and `https://github.com/uzh/signal-collect-graphs` at revision 0149927e68. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldEFORUhISkJVSy15Nmd6QnlqYzFKWUE`.

while the asynchronous scheduling found a solution within a few seconds. One explanation is that with a synchronous scheduling the vertices tend to switch states in lockstep, which has them cycle through or oscillate between conflicts (“thrashing”). The results show that for some algorithms asynchronous scheduling can be crucial for fast convergence. Other algorithms share this property: Koller and Friedman note that some asynchronous loopy belief propagation computations converge where the synchronous computations keep oscillating. They summarise in that context that [31, p. 408]: *“In practice an asynchronous message passing scheduling works significantly better than the synchronous approach."*

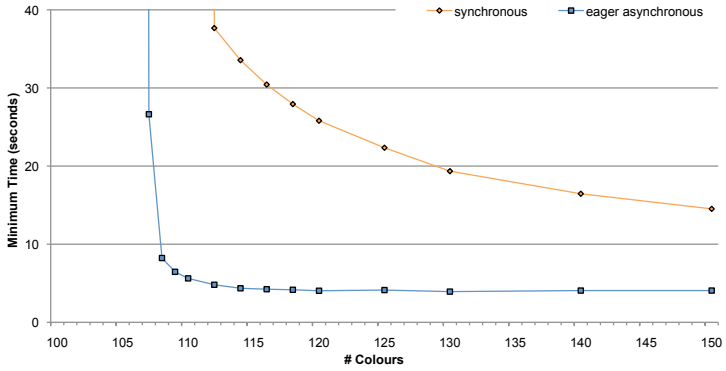


Fig. 19. Vertex colouring with a varying number of colours on a Latin Square problem with 100 * 100 vertices and almost two million directed edges.

6 Related Work

In this section we give an overview over the foundations of SIGNAL/COLLECT and alternative approaches to large-scale graph processing.

6.1 Foundations

The SIGNAL/COLLECT programming model is related to three lower-level programming models, which are all suitable for distributed and parallel computations, but lack a focus on graph computations:

– *Bulk-synchronous parallel (BSP)*
  In BSP [57], a parallel computation consists of a sequence of supersteps. During each superstep, components process some assigned task and communicate with each other. There is a periodic global synchronisation that ensures that all tasks of the superstep have been completed before the next superstep is started. The synchronous scheduling of SIGNAL/COLLECT is an implementation of this model.

– *Actor model*
  In the actor programming model [22], many processing components take part in a computation and operate in parallel. These components can only influence each other via asynchronous message passing. The asynchronous scheduler of SIGNAL/COLLECT was inspired by the actor model.

– *Data-flow model*
  Depending on the context, the expression "data-flow" can have different meanings. We understand it broadly as a programming model where a computation is defined as a dependency graph in which data flows along edges and vertices use their input data to compute new data that gets sent along their outgoing edges. This model can be seen as a specialisation of the actor model, where each vertex is represented by an actor and communicates along the graph structure.

When designing a programming model for graph processing, it is important to consider the different kinds of computations on graphs. There are two fundamentally different ways of thinking about computations on graphs. One way to interpret a graph is as a data structure, where data can be associated with vertices and edges. Computations may explore this structure and modify its data, potentially iteratively, until some termination or convergence criterium is reached. We refer to computations with this characteristic as *data-graph* computations. Another way to interpret a graph is as a plan that determines the flow of data along processing stages. Vertices represent processing stages for data, while edges represent the (potentially cyclic) paths along which data flows. This view encompasses the *data-flow* programming model.

With SIGNAL/COLLECT, we have designed a programming model that is suitable for both kinds of computations: In the SIGNAL/COLLECT

programming model vertices are processing units akin to actors whilst edges can have associated data and may compute signals that flow to their target vertex.

Researchers with roots in disparate communities such as machine learning, biology, or the Semantic Web have answered the call for general programming models and frameworks specialised for scalable graph processing.

Figure 20 provides a high-level overview of distributed data processing systems that support iterated processing. It differentiates systems along their ability for synchronous versus asynchronous processing of the data on the y-axis and the kind of data abstraction they operate on (key-value pairs, sets, or tables versus graphs) on the x-axis. The category "Synchronisation Required" encompasses systems that schedule iterated computations with mandatory global barriers between iterations. The "Asynchronous Possible" category encompasses systems that are able to schedule iterated computation without such global barriers. "MR Based" is used as a category label for systems that extend the MapReduce model with support for iterated processing or graph abstractions.

We focus our discussion on programming models and systems that are geared towards processing graphs, especially ones that are capable of asynchronous processing. Specifically, we first present GraphStep and Pregel, which have inspired many other graph specialised BSP-based systems. After that, we discuss GraphLab, its extension PowerGraph, and HipG in detail, because they are vertex centric approaches that are closely related to SIGNAL/COLLECT. In the last part we give summaries of other related work.

6.2 GraphStep and Pregel

GraphStep [12] is the programming model that is most closely related to SIGNAL/COLLECT. It combines the BSP programming model with the concept of vertices as actors and edges representing the communication structure between those actors. A computation progresses with all vertices receiving input messages, awaiting a global synchronisation, performing a local update operation, and sending output messages. The last two steps broadly correspond to collecting and then signalling. A newer description of the model [11] adds a reduce phase on the incom-

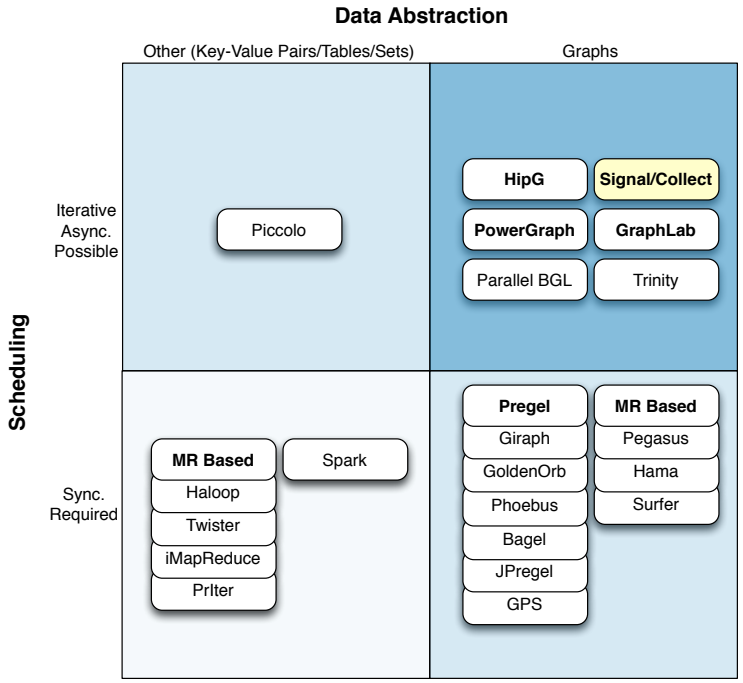


Fig. 20. Selected Related Work: An overview of only the high-level distributed data processing *systems* that support iterated processing.

ing messages and a separate edge function where each edge reads and writes local state and then possibly sends a message to its destination vertex. The model is meant to be implemented in field-programmable gate array (FPGA) circuits and does not support data-flow computations, asynchronicity, or graph modifications.

Pregel is a framework with a similar programming model developed by Google for large-scale graph processing [41]. The framework scales to graphs with billions of vertices and edges via distribution to thousands of commodity PCs. Pregel is based on a programming model that was inspired by BSP: A computation consists of a sequence of supersteps. During a superstep, each vertex executes an algorithm-specific compute function that can modify the vertex state, modify the graph, and send messages to other vertices. Global synchronisations ensure that all compute functions of the superstep have been completed before the next superstep is started. Within a compute function, a vertex can vote to halt the computation. A computation ends when all the vertices have voted to

halt. In order to reduce the number of messages that are sent between workers/machines, Pregel supports combiners that aggregate multiple messages for the same vertex into one. For the computation of global values Pregel also supports aggregation operations.

The Pregel model merges computation, communication, and termination detection into one compute function on a vertex. This function is a black box from the perspective of the framework, which requires a manual implementation of termination detection and prevents the scheduler from separately scheduling state updates and communication.

Pregel can only handle synchronous computations. In a synchronous computation one problematic operation or node can be enough to slow all computations, while in an asynchronous computation only operations on that node or the specific operation are slowed. As we discussed in our evaluation (see Section 5.4), synchronous scheduling can also lead to convergence problems due to oscillations for some algorithms.

Pregel supports graphs with one kind of vertex type sharing a single compute function.This complicates the reusability of vertex/edge-specifications and it adds complexity when implementing algorithms with multiple kinds of vertices or edges. These constraints also make it harder to compose several algorithms within the same computation.

There are extensions to the model for incremental recomputations [5] and for custom scheduling that can imitate some of the properties of an asynchronous scheduling [58].

Google has not released its implementation of Pregel, but there exist several related open source implementations such as Giraph[19], GoldenOrb[20], Phoebus[21], Bagel[22], and JPregel[23]. In addition, two Pregel-like implementations were developed with a special focus: Menthor[24] is an open source implementation associated with research into high-level control structures over computation steps [21] and the Graph Processing System (GPS) implementation supports static and dynamic graph partitioning [50]. Also noteworthy is Green-Marl [24], a domain-specific

[19] http://giraph.apache.org/
[20] http://goldenorbos.org/
[21] https://github.com/xslogic/phoebus
[22] https://github.com/mesos/spark/blob/master/bagel
[23] http://kowshik.github.com/JPregel/
[24] http://lcavwww.epfl.ch/~hmiller/menthor/

programming language for graph analysis algorithms that compiles to execution systems with a Pregel-like programming model.

6.3 GraphLab

GraphLab is a programming model and framework for parallel graph algorithms [36]. The programming model is especially suitable for computations with sparse data dependencies and for asynchronous iterative computation.

GraphLab is based on a data-graph model which simultaneously represents data and computational dependencies. A computation progresses by executing update functions on vertices. These functions can modify the vertex and edge data as well as data associated with neighbouring vertices in the data-graph. The model offers flexible scheduling of these update operations as well as functions to aggregate over the state of the entire data-graph. The scheduler supports different consistency guarantees, which permit the adaptation of some algorithmic correctness proofs from a sequential to a parallel setting.

In full-consistency mode, concurrent modifications to the neighbourhood of a vertex have to be prevented while an update function is executed. Assuming a random distribution of vertices over cluster nodes of a large cluster—an assumption currently true for many frameworks such as HipG (see below) and Pregel—the expected (and worst-case) scenario is that the vertex data and the data of neighbouring vertices are spread over almost as many nodes as there are vertices in the neighbourhood. The authors describe in a more recent publication [16] (see below) that executing an update function in full-consistency mode on a vertex with a sizeable neighbourhood is a costly operation, because it requires a distributed lock of a large fraction of the cluster. They also mention that "distributed locking and synchronisation introduces substantial latency" [37]. Furthermore, they describe that "the locking scheme used by GraphLab is unfair to high degree vertices" (see [16], Section 4.3.2).

Another scheduler ("chromatic engine") works around some of these issues in distributed computations [37]. This scheduler uses a vertex colouring to avoid the expensive locking during execution. It is equivalent to a BSP execution where at each step only vertices with the same colour are active. This scheduler requires finding a graph colouring (more constrained ones for strong consistency guarantees) and the number of

processing steps and global synchronisations is multiplied by the number of colours used for the graph colouring. There seems to be a trade-off between the availability of consistency guarantees and the effort of obtaining a graph colouring.

GraphLab does currently not allow graph modifications during a computation and does not support graphs with multiple vertex types, which complicates composition of algorithms and reusability of components. Lastly, GraphLab has undirected edges. Hence, algorithms that exploit directionality would have to encode it in an edge's data, complicating the framework's ability to optimise computations based on directionality.

6.4 PowerGraph

PowerGraph [16] is a substantial redesign and reimplementation of GraphLab. The main difference in PowerGraph's abstraction lies in the computation's division into three phases: gather roughly corresponds to a Pregel combiner gathering and aggregating the data from neighbours, apply computes the new vertex state, and scatter updates the values of the adjacent edges. According to the execution semantics ([16], Algorithm 1, Section 4.1) the three functions are always called sequentially, without interruption, possibly complicating scheduler-based optimisations. Akin to GraphLab, PowerGraph does not allow for multiple implementations of the three functions per graph and it does not support graph modifications during computations.

To enable a more efficient implementation of the distribution PowerGraph introduces the idea of vertex cuts – essentially the replication of vertices to many machines. Whilst this reduces cross-machine communication for some algorithms and more evenly distributes the load of high-degree vertices, it introduces replication of the vertices and their associated state/data up to a factor of 5-18 for 64 machines. The variation of the overhead factor is dependent on the partitioning strategy chosen. Smarter strategies reduce the replication factor but increase graph loading time by a factor of about 5 (when using 64 machines).

In a direct comparison of the programming models, PowerGraph has consistency guarantees and the built-in optimisations for high-degree vertices going for it. SIGNAL/COLLECT offers more flexibility with regard to the efficient edge representation, and can even route messages to

another vertex if there is no explicitly stored edge between them. SIGNAL/COLLECT also has built-in support for using different vertex, edge and message types inside the same graph, whilst the same requires a custom mapping and is potentially inefficient for PowerGraph.

PowerGraph's lack of support for modifications during a computation or the efficiency of the asynchronous implementation are most likely engineering related and not fundamental properties of the approach.

6.5 HipG

HipG is a distributed framework that facilitates high-level programming of parallel graph algorithms by expressing them as a hierarchy of distributed computations [32, 33].

As in the Pregel model, code is executed on a vertex. But while in Pregel messages are sent to other vertices, a HipG vertex can conceptually directly execute functions on neighbouring vertices (the framework translates those function calls to asynchronous messages). HipG supports synchronisers which are coordinators for function executions that have the option to block until all executed functions have completed. This feature can also be used to aggregate global values. A synchroniser can spawn additional synchronisers to create hierarchical computations. This is especially useful for divide and conquer algorithms on graphs.

While it is possible to write a compute function for a vertex that handles thousands of received messages at once, there is no obvious way of combining functions, which means that they all have to be executed. This could be problematic if one wants to implement an iterated computation, because it would require for a function to spawn as many new functions as there are neighbours, potentially leading to an exponential growth of functions in the system. One solution is to use a global synchroniser that repeatedly executes functions on all vertices (using a "visit" flag and only propagating onwards if the flag is not set yet) and has barriers (synchronisations) between those executions (indeed, this is how the PageRank is implemented in the example code provided with the system[25]) – an implementation of a BSP-scheduler with HipG primitives.

[25] `http://www.few.vu.nl/~e.krepska/HipG/`

6.6 Other Related Work

The Parallel BGL[26] is a generic C++ library of parallel and distributed graph algorithms and data structures [38, 18, 19]. One of the main design goals of this system (and also of the ParGraph[27] system) was to allow for sequential BGL[28] algorithms to be "lifted" to parallel programs whilst minimising the required changes. It has support for a special process group that delivers messages immediately instead of waiting for a BSP step synchronisation, but this feature is not explored in any depth.

Najork et al. evaluated three different platforms and programming models on large graphs [43]. The focus of the evaluation is to find the trade-offs between the platforms for various algorithms and did not include vertex-centric models. Members of the same lab are also working on the Trinity graph engine, a distributed key-value store with optimisations for vertex-centric graph processing, such as bundling of messages, graph partitioning and low latency processing [61]. Trinity also supports asynchronous processing and while the technology and features of the framework are impressive, it is not publicly available and the report gives little information about the properties of the programming model and about how algorithms are expressed.

There are several systems for large-scale graph computations implemented on top of MapReduce or by generalising the MapReduce model. Most of these systems have limited support for iterated computations and do not support asynchronicity [52, 28, 8, 20]. PrIter is a modified version of Hadoop MapReduce that supports executing processing steps only on a subset of items with priorities above a threshold [63]. Kajdanowicz et al. compared the efficiency of a MapReduce-based system with a BSP-based system for processing large graphs and conclude that BSP can outperform MapReduce by up to an order of magnitude [27].

Piccolo and Spark are distributed processing platforms that use table-/set-based abstractions, support iteration and can serve as the foundation of more specialised graph processing frameworks [48, 60]. One such extension is the aforementioned Bagel which is built on Spark.

Also noteworthy are distributed data-flow engines such as Sawzall [47], Dryad/DryadLINQ [26, 25], Pig [45], and Ciel/Skywriting [42].

[26] Parallel Boost Graph Library: `http://osl.iu.edu/research/pbgl/`
[27] `http://pargraph.sourceforge.net/`
[28] `http://www.boost.org/doc/libs/1_46_1/libs/graph/doc/index.html`

Computations on graphs require a custom mapping to the respective dataflow language model. Some of the languages allow to express iterated processing, but the underlying systems are not optimised for doing this efficiently on graph structured data.

There are more graph processing libraries that focus on specific algorithms, but did not offer a detailed enough explanation of a more general programming model. Also we did not cover frameworks that focus on specific aspects of scaling algorithms on architectures such as supercomputers or GPUs, because the scalability challenges are different.

7 Limitations and Future Work

As we have seen in previous sections, SIGNAL/COLLECT is a scalable programming model and framework for parallel and distributed graph algorithms. Whilst our expressiveness evaluation is limited by the number of algorithms shown, their wide variability indicates some generalisability of our claim of simplicity and expressiveness. The generalisability of our scalability evaluations is also limited by the number of algorithms tested. But again, we believe that the range of algorithms is typical for such evaluations and provides a strong indication for SIGNAL/COLLECT's scalability.

In addition, our evaluation does raise some very interesting questions: Could we improve performance with a better graph partitioning scheme? How does SIGNAL/COLLECT fare with graphs that contain vertices with many incoming/outgoing edges? How could SIGNAL/COLLECT recover from failures? And, what would the impact of a prioritising scheduler be on run-time performance? We discuss these questions in the remainder of this section.

Due to the default partitioning scheme the vast majority of signals in the distributed version are sent over the network. This is inefficient, but it could be improved without modifying the programming model: A domain optimised hash function could be used, for example one that maps websites from the same domain to the same worker or cluster node. This should improve locality of signalling. It would be interesting to see to what degree such a scheme would suffer from imbalanced loads for different domains.

We did not encounter any problems due to high in/out-degree vertices so far. Whilst Pregel-style combiners address the problem of *high*

in-degree vertices, the problem of *high out-degree vertices* could be addressed by modifying a graph: High out-degree vertices could create child vertices that each inherit a share of the outgoing edges. All state changes and further edge additions/removals are forwarded to the child vertices. High out-degree child vertices could recursively use the same scheme. A more efficient and more limited alternative is to parallelise the signalling on a vertex to "smear" the signalling workload across a cluster node instead of having the entire load on one worker.

All our experiments were run on either web-style real-world graphs or the synthetic Kronecker graphs. These graphs have an uneven degree distribution [34]. Consequently, one might argue that our findings may have a limited generalisability. As discussed above, however, we believe that SIGNAL/COLLECT could be adapted to process high-degree vertices. We do acknowledge, that the memory overhead of representing graphs as collections of edges and vertices may rise above matrix-based representations in highly connect graphs that would result in non-sparse matrices. Note however, that such graphs seem uncommon on the (Semantic) Web, for social networks, and for many naturally occurring phenomena, as these have been found to follow a power law degree distribution [34].

SIGNAL/COLLECT currently only supports very primitive checkpointing and no error recovery. All the jobs that we have run so far were very short-lived and we never encountered hardware failures. Hence, for us it would make more sense to simply restart a failed job. With the long-running jobs and use cases such as query processing error recovery would become more important. The distributed snapshot algorithm [7] would probably be a good candidate for addressing this issue, because it could run without interrupting algorithm execution.

Finally, it would be interesting to experiment with a prioritising scheduler. Such a scheduler might have benefits for use cases in which computing and sending signals is very expensive relative to other tasks. Otherwise, the overhead of prioritising operations may not pay off.

8 Conclusions

Both researchers and industry are confronted with the need to process increasingly large amounts of data, much of which has a natural graph representation. In order to address the need to run algorithms on increasingly large graphs we have designed a programming model that is both

simple and expressive. We showed its expressiveness by designing adaptations of many interesting algorithms to the programming model and the simplicity by being able to express these algorithms with just a few lines of code.

We built an open source framework that can parallelise and distribute the execution of algorithms formulated in the model. We empirically evaluated the scalability of the framework across different graph structures and algorithms and have shown that the framework scales with additional resources. The framework offers great efficiency and performance on a cluster architecture, which was shown by loading the huge Yahoo! AltaVista webgraph and computing high-quality PageRanks for its vertices in just 3 minutes on a dozen machines.

With SIGNAL/COLLECT we have created a programming abstraction that allows programmers to run algorithms quickly on large graphs without worrying about the specifics of how parallel and distributed processing resources are allocated. We believe that marrying actors with a graph abstraction should be taken beyond simple graph processing and that this is an effective approach to building dynamic and complex systems that operate on large data sets.

Acknowledgements We thank the Hasler foundation for funding our research and Yahoo! for giving us access to the AltaVista Web Page Hyperlink Connectivity Graph. We thank Mihaela Verman, Lorenz Fischer, and Patrick Minder for the many interesting discussions, for feedback, and proofreading, as well as William Cohen for input on earlier versions of the project. We thank Francisco de Freitas for designing the first prototype of an Akka-based distributed version of the SIGNAL/COLLECT framework. We thank the reviewers Jacopo Urbani, Michael Granitzer, and Sang-Goo Lee for their feedback and suggestions for improvements.

References

[1] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM. ISBN 1-58113-840-7. doi: 10.1145/1007912.1007931. URL http://doi.acm.org/10.1145/1007912.1007931.

[2] David A Bader and Kamesh Madduri. Snap, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[3] Chris Biemann. Chinese whispers: an efficient graph clustering algorithm and its application to natural language processing problems. In *Proceedings of the First Workshop on Graph Based Methods for Natural Language Processing*, TextGraphs-1, pages 73–80, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1654758.1654774.

[4] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010. ISSN 2150-8097.

[5] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *Proceedings of the fourth international workshop on Cloud data management*, CloudDB '12, pages 1–8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1708-5. doi: 10.1145/2390021.2390023. URL http://doi.acm.org/10.1145/2390021.2390023.

[6] Soumen Chakrabarti, Byron Dom, and Piotr Indyk. Enhanced hypertext categorization using hyperlinks. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 307–318, New York, NY, USA, 1998. ACM. ISBN 0-89791-995-5. doi: 10.1145/276304.276332. URL http://doi.acm.org/10.1145/276304.276332.

[7] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985. ISSN 0734-2071. doi: 10.1145/214451.214456. URL http://doi.acm.org/10.1145/214451.214456.

[8] Rishan Chen, Xuetian Weng, Bingsheng He, and Mao Yang. Large graph processing in the cloud. In *SIGMOD Conference'10*, pages 1123–1126, 2010.

[9] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29 –41, july-aug. 2009. ISSN 1521-9615. doi: 10.1109/MCSE.2009.120.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1251254.1251264.

[11] Michael deLorimier. *GRAph Parallel Actor Language âĂŤ A Programming Language for Parallel Graph Algorithms*. PhD thesis, California Institute of Technology, 2013. http://resolver.caltech.edu/CaltechTHESIS:08192012-145253489.

[12] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, TomÃąs E. Uribe, Thomas F. Knight, and AndrÃl' Dehon. Graphstep: A system architecture for sparse-graph algorithms. In *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE*. IEEE Computer Society, 2006.

[13] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In Salim Hariri and Kate Keahey, editors, *HPDC*, pages 810–818. ACM, 2010. ISBN 978-1-60558-942-8.

[14] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, pages 120–123, October 1970.

[15] Lise Getoor, Eran Segal, Ben Taskar, and Daphne Koller. Probabilistic models of text and link structure for hypertext classification. In *In IJCAI Workshop on Text Learning: Beyond Supervision*, 2001.

[16] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, Berkeley, CA, USA, 2012. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387883.

[17] Mark Granovetter. Threshold Models of Collective Behavior. *American Journal of Sociology*, 83(6):1420–1443, 1978. ISSN 00029602. doi: 10.2307/2778111. URL http://dx.doi.org/10.2307/2778111.

[18] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[19] Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not.*, 40:423–437, October 2005. ISSN 0362-1340.

[20] Yunhong Gu, Li Lu, R. Grossman, and A. Yoo. Processing massive sized graphs using sector/sphere. In *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*, pages 1 –10, nov. 2010. doi: 10.1109/MTAGS.2010.5699427.

[21] P. Haller and H. Miller. Parallelizing machine learning– functionally: A framework and abstractions for parallel graph processing. In *the 2nd Annual Scala Workshop 2011*, June 2011.

[22] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[23] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, April 2011. ISSN 1095-9203. doi: 10.1126/science.1200970. URL http://dx.doi.org/10.1126/science.1200970.

[24] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying scalable graph processing with a domain-specific language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 208. ACM, 2014.

[25] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 987–994, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559962. URL http://doi.acm.org/10.1145/1559845.1559962.

[26] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: http://doi.acm.org/10.1145/1272996.1273005.

[27] Tomasz Kajdanowicz, Wojciech Indyk, Przemyslaw Kazienko, and Jakub Kukul. Comparison of the efficiency of mapreduce and bulk synchronous parallel approaches to large network processing. In *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*, pages 218 –225, dec. 2012. doi: 10.1109/ICDMW.2012.135.

[28] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system. *Data Mining, IEEE International Conference on*, 0:229–238, 2009. ISSN 1550-4786.

[29] U Kang, D.H. Chau, and C. Faloutsos. Inference of beliefs on billion-scale graphs. *The 2nd Workshop on Large-scale Data Mining: Theory and Applications*, 2010.

[30] Christoph Kiefer, Abraham Bernstein, and André Locher. Adding data mining support to sparql via statistical relational learning methods. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 478–492. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-68233-2. doi: 10.1007/978-3-540-68234-9_36. URL http://dx.doi.org/10.1007/978-3-540-68234-9_36.

[31] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Jan 2009.

[32] Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, and Henri Bal. A high-level framework for distributed processing of large-scale graphs. In *Proceedings of the 12th international conference on Dis-*

tributed computing and networking, ICDCN'11, pages 155–166, Berlin, Heidelberg, 2011. Springer-Verlag.

[33] Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, and Henri Bal. Hipg: parallel processing of large-scale graphs. *SIGOPS Oper. Syst. Rev.*, 45(2):3–13, July 2011. ISSN 0163-5980. doi: 10.1145/2007183.2007185. URL http://doi.acm.org/10.1145/2007183.2007185.

[34] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *The Journal of Machine Learning Research*, 11:985–1042, 2010.

[35] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0214-2.

[36] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[37] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Graphlab: A distributed framework for machine learning in the cloud. *CoRR*, abs/1107.0922, 2011.

[38] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, pages 5–20, 2007.

[39] Sofus A. Macskassy and Foster Provost. A simple relational classifier. In *Proceedings of the Second Workshop on Multi-Relational Data Mining (MRDM-2003) at KDD-2003*, pages 64–76, 2003.

[40] Sofus A. Macskassy and Foster Provost. Classification in networked data: A toolkit and a univariate case study. *J. Mach. Learn. Res.*, 8:935–983, May 2007. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1248659.1248693.

[41] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *SIGMOD*. ACM, 2010. ISBN 978-1-4503-0032-2.

[42] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a uni-

versal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1972457.1972470.

[43] Marc Najork, Dennis Fetterly, Alan Halverson, Krishnaram Kenthapadi, and Sreenivas Gollapudi. Of hammers and nails: an empirical comparison of three paradigms for processing large graphs. In *Proceedings of the fifth ACM international conference on Web search and data mining*, WSDM '12, pages 103–112, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0747-5. doi: 10.1145/2124295.2124310. URL http://doi.acm.org/10.1145/2124295.2124310.

[44] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559911. URL http://doi.acm.org/10.1145/1559845.1559911.

[45] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6.

[46] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.1768.

[47] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, October 2005. ISSN 1058-9244. URL http://dl.acm.org/citation.cfm?id=1239655.1239658.

[48] Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX

Association. URL http://portal.acm.org/citation.cfm?id=1924964.

[49] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 2nd international edition edition, 2003. ISBN 9788129700414. URL http://aima.cs.berkeley.edu.

[50] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. Technical report, Stanford, http://infolab.stanford.edu/gps/publications/tech-report.pdf, 2012.

[51] Stefan Schurgast. Markov logic inference on signal/collect. Master's thesis, University of Zurich, Department of Informatics, 2010.

[52] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. *Cloud Computing Technology and Science, IEEE International Conference on*, 0:721–726, 2010.

[53] Daniel Strebel. Making signal/collect scale. B.s. thesis, University of Zurich, 2011.

[54] Philip Stutz, Abraham Bernstein, and William W. Cohen. Signal/-Collect: Graph Algorithms for the (Semantic) Web. In *ISWC*, 2010.

[55] Philip Stutz, Mihaela Verman, Lorenz Fischer, and Abraham Bernstein. Triplerush: A fast and scalable triple store. In Thorsten Liebig and Achille Fokoue, editors, *SSWS@ISWC*, volume 1046 of *CEUR Workshop Proceedings*, pages 50–65. CEUR-WS.org, 2013.

[56] Philip Stutz, Daniel Strebel, and Abraham Bernstein. Signal/collect: Processing large graphs in seconds. *Semantic Web*, 7(2):139–166, 2016. ISSN 1570-0844. doi: 10.3233/SW-150176.

[57] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL http://doi.acm.org/10.1145/79173.79181.

[58] Guozhang Wang, Wenlei Xie, Alan Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[59] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proc. of the 34th Intl Conf. on Very Large Data Bases (VLDB)*, February 2008.

[60] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

[61] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. *Proceedings of the VLDB Endowment*, 6(4), 2013.

[62] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. imapreduce: A distributed computing framework for iterative computation. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1112–1121, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4577-6. doi: 10.1109/IPDPS.2011.260. URL http://dx.doi.org/10.1109/IPDPS.2011.260.

[63] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 13:1–13:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038929. URL http://doi.acm.org/10.1145/2038916.2038929.

[64] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, 2002.

Distributed TripleRush

An Asynchronous Graph Store Architecture

This chapter is based on an unpublished paper, which was in turn a significant extension of the ideas described in [15]. A modified version of the paper extended with "random walks with restarts" was later published at WWW [16]. The co-authors have kindly agreed to permit inclusion as part of this thesis.

Distributed TripleRush

An Asynchronous Graph Store Architecture

Philip Stutz, Bibek Paudel, Mihaela Verman, and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
philip@stutz.tech, {bpaudel, verman, bernstein}@ifi.uzh.ch

Abstract. In this paper we propose to rethink query execution within triple stores as a highly parallelised graph exploration, where threads asynchronously explore binding candidates. To evaluate this architecture we implemented *d-TripleRush*, which is built on a distributed graph processing system and processes queries by routing query descriptions through an active data structure. In experiments we find that our architecture can be used to build a competitive graph store that exploits parallelism where possible and leverages the asynchronous nature of its graph exploration to provide first results quickly. We also evaluate the scalability and show that this architecture can support fast answer times even on datasets with more than a billion triples.

1 Introduction

Use cases such as social network analysis, monitoring of financial transactions or analysis of web pages and their links require large-scale graph computation. To address this need, many have researched the development of efficient triple stores [1, 18, 10]. These systems borrow from the database literature to investigate efficient means for storing large graphs and retrieving subgraphs, which are usually defined via a pattern matching language such as SPARQL. Even though these systems process graphs, most of them leverage decades of research results in efficient processing of partial answer-sets by mapping the graphs into set-/array-style internal data structures. They are built like a centralised database, raising the question of scalability and parallelism within query execution.

To increase the parallelism of such graph stores, modern solutions propose the use of parallel operators [19], sideways information-passing [11], or even pipelined operations and replication [5]. Other approaches focus on building triple stores based on specialised programming models for distributed systems: MapReduce [3] has been used to aggregate

results from multiple single-node RDF stores in order to support distributed query processing [6] or to process whole SPARQL query execution pipelines (e.g., [7]).

In this paper we propose to rethink query execution within graph stores in the light of the changes of computer architectures. *We propose to exploit the large number of execution units of modern servers via the parallel exploration of partial bindings*. Specifically, we propose to explore each partial binding to a query in parallel in a graph exploration style forking the exploration whenever more than one binding is possible, returning the result when all variables of an exploration are bound, and expiring the exploration when it reaches a dead end.

We implemented *distributed TripleRush*[1] (d-TripleRush for brevity) to explore this architecture. d-TripleRush is built on the distributed graph processing system SIGNAL/COLLECT [14].[2] d-TripleRush is a significant redesign and extension of TripleRush [15], which was limited to parallelising queries on a single machine. Whilst traditional stores pipe data through query processing operators, d-TripleRush asynchronously routes query descriptions through an active data structure. For this reason, d-TripleRush does not use any joins in the traditional sense, but searches the index graph in parallel.

In the following, we describe the novel distributed architecture, as well as the functionality and interactions of its building blocks. We then compare the architecture with traditional graph store approaches. Next, we evaluate the approach on multiple benchmarks and show that it can offer competitive performance, as well as good scalability. We close with a discussion of the limitations.

2 Related work

Studies related to d-TripleRush can be divided into three categories: (i) distributed in-memory triple-stores, (ii) graph computation frameworks, and (iii) studies into RDF index structures.

Distributed in-memory triple stores: Most closely related to d-TripleRush are Trinity.RDF [19] and TriAD [5]. Trinity.RDF is also built on top of a distributed graph processing abstraction and is based on the distributed

[1] `https://github.com/uzh/triplerush` (Apache 2.0)
[2] similar to Pregel [9], GraphLab/PowerGraph [4], Trinity [12]

Bulk-Synchronous approach [17]. It uses parallel operators to improve SPARQL performance. TriAD's approach focuses on pre-processing, pipelined operations, and replication of indices. These systems all strive for efficient and scalable execution of SPARQL queries, but the asynchronous querying approach of d-TripleRush, where many copies of query descriptions are routed through a specialised index structure, is fundamentally different.

Distributed graph computation frameworks: A number of distributed graph processing frameworks have been proposed in recent years [8, 14, 4]. Whilst these systems provide a basis for building distributed analytic solutions, they do not provide a high-level graph querying language such as a SPARQL.

RDF index structures: d-TripleRush builds on the insights into RDF indexing in the past years [10, 18] in that it builds a multi-level structure of increasingly specific nodes. It differs significantly from these investigations in that the index structure is optimised for a query execution that is based on highly parallelised asynchronous routing of partially bound results.

3 d-TripleRush Architecture

The core idea of d-TripleRush is to build a triple store with three types of SIGNAL/COLLECT vertices: Each *index vertex* corresponds to a triple pattern, each *triple vertex* corresponds to an RDF triple, and *query vertices* coordinate query execution. Partially matched copies of queries are routed in parallel along different paths of this structure. The index graph is, therefore, optimised for efficient routing of query descriptions to data and its vertices are addressable by an ID, which is a unique [subject predicate object] tuple.

We first describe how the graph is conceptually built and then explain the details of how this structure enables efficient parallel graph exploration.

3.1 Building the Index Graph

d-TripleRush is a triple store with three types of SIGNAL/COLLECT vertices:

Triple vertices (level 4, Fig. 1) represent triples in the database. Each contains subject, predicate, and object information.

Index vertices (levels 1-3, Fig. 1) represent triple patterns and are responsible for routing partially matched copies of queries (referred to as *query particles*) towards triple vertices that match their respective patterns. They also contain subject, predicate, and object information, but one or several of them are wildcards.

Query vertices (Fig. 2) are added to the graph for each query that is being executed. The query vertex emits the first query particle that traverses the index structure. Once all query particles—successfully matched or not—get routed back to their respective query vertex, it reports the results and removes itself from the graph.

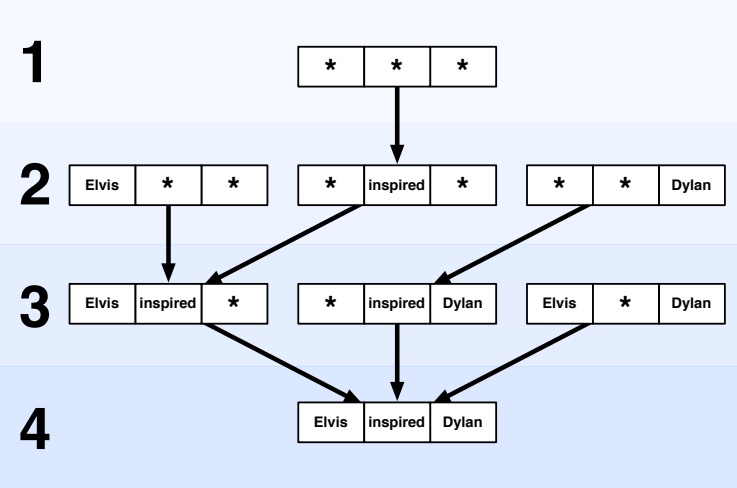


Fig. 1. d-TripleRush index graph for the triple vertex [Elvis inspired Dylan].

The graph is built bottom-up, starting by creating a *triple vertex* for each RDF triple. These vertices are added to SIGNAL/COLLECT, which turns them into parallel processing units. A triple vertex will add its immediate *index vertices* (if they do not exist yet) and an edge from these vertices to itself. The construction process continues recursively for the index vertices until the parent vertex has already been added or the index vertex has no parent.

The index structure illustrated in Fig. 1 ensures that there is exactly one path from an index vertex to each triple vertex below it.
Observations: The number of predicates is usually much smaller than the number of distinct subjects or objects. Hence, storing edges from the root to [* P *] vertices requires the least amount of memory. The index graph we just described is different from traditional index structures, because it is designed for the efficient parallel routing of messages to triples corresponding to a given triple pattern. All vertices that form the index structure are active parallel processing elements that only interact via message passing.

3.2 Query Execution

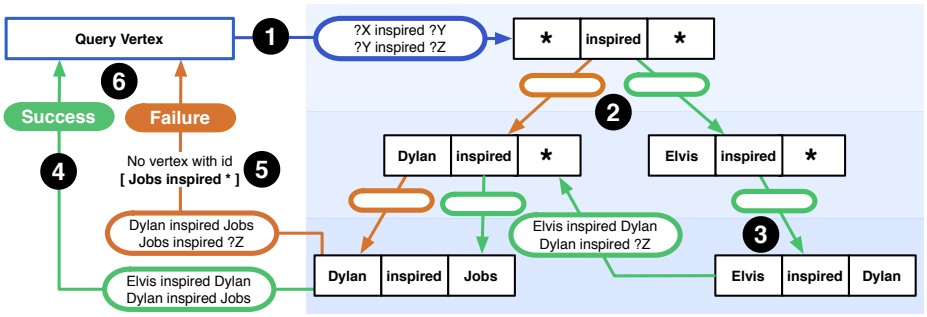


Fig. 2. Query execution on the relevant part of the index that was created for the triples [Elvis inspired Dylan] and [Dylan inspired Jobs].

Consider the subgraph shown in Fig. 2 and the query processing for the query: (unmatched = [?X inspired ?Y], [?Y inspired ?Z]; bindings = {}). The query execution starts by adding the query vertex to the TripleRush graph. After invoking a standard SPARQL query optimizer [13] that reorders the triple patterns for optimal execution the query gets processed as follows:

1. The query vertex emits a single query particle, which is routed (by SIGNAL/COLLECT) to the index vertex that matches its first unmatched triple pattern. To determine when a query has finished processing, the initial query particle is endowed with a large number of tickets (Long.MaxValue). If the tickets should run out that is detected and

there would be various ways to deal with this (by acquiring new tickets from the query vertex or by assigning more tickets initially).

2. When a query particle arrives at an index vertex, a copy of it is sent along each edge. The original particle evenly splits up its tickets among its copies.
3. Once a query particle reaches a triple vertex, the vertex attempts to match the next unmatched query pattern to its triple. If this succeeds, then a variable binding is created and the remaining triple patterns are updated with the new binding. The query particle gets sent to the index or triple vertex that matches its next unmatched triple pattern.
4. If all triple patterns are matched, then the query particle gets routed back to its query vertex.
5. If no vertex with a matching pattern is found, then a handler for undeliverable messages routes the failed query particle back to its query vertex.
6. Query execution finishes when the sum of tickets of all failed and successful query particles received by the query vertex equals the initial ticket endowment of the first particle that was sent out. The query vertex reports that all results have been delivered and removes itself from the graph.

Observations: Queries are often routed along downward edges in the index structure, and placing the index vertices in a way that achieves good locality, means that few messages are sent across machines. We found that the following scheme can achieve good locality, while at the same time ensuring a high degree of parallelism: If the subject of an index vertex is defined, then it is placed on a node determined by its subject. If the subject is a wildcard, then it is placed on a node determined by the object. If only the predicate is defined, then it is placed on a node determined by the predicate. The root index vertex is hardcoded to the last node. This scheme guarantees that particles are locally routed from [S * *] to [S P *] as well as from [* * O] to [* P O].

In addition, to assign a vertex to workers on a machine identified with the above assignment scheme, we compute the sum of its (encoded, see next section) IDs modulo the number of workers on the assigned node. In our tests, this scheme performed better than mixing the values with a collision-minimising hash function. Signal/Collect uses the same mappings for vertex addressing and for routing messages to (potentially non-existent) index vertices.

3.3 d-TripleRush Optimisations

Just like parallel TripleRush [15], d-TripleRush contains some initial optimisations: a) we do dictionary encoding, b) we remove the triple vertices and fold them into the third index level, where each index vertex stores a compact representation of all the triples that match their pattern, c) we only send the tickets of the failed particles back to the query vertex, and d) we use bulk-messaging and message-combiners.

In addition to this, d-TripleRush contains improvements that address previous limitations with regard to memory usage during loading, and insert performance by adopting a new data structure for the index vertices. Next, we discuss the details and motivation of these changes.

Index Vertex Representation: In Fig. 1, one notices that the id of an index vertex varies only in one position—the subject, the predicate, or the object—from the ids of its children. To reduce the size of the edge representations, we do not store the entire id of child vertices, but only the specification of this position consisting of one dictionary encoded number per child. We refer to these numbers as *id-refinements*. The same reasoning applies to binding index vertices, where the triples they store only vary in one position from the id of the binding index vertex.

Routing and binding only require a traversal of all id-refinements. To support traversal and inserts in a memory-efficient way, we store the refinements in a special-tailored Splay tree, where the key of each node is an interval and each node stores the set of refinements contained in its interval in a sorted array with delta-encoding. The data structure supports low memory usage, fast traversal, and amortized $O(log(n))$ worst case time complexity inserts, where n is the number of dictionary encoded items.

Index Graph Structure: Because we fold the triple vertices into the third index level, there is no longer an obvious place where one can verify if a fully bound pattern corresponds to a triple that exists inside the store.

To deal with this, d-TripleRush sends the particle that has to check for the existence of a fully-bound pattern to the corresponding [S * O] index vertex. These vertices do not store the ID-refinements into a Splay-tree, but into a sorted array. The existence is thus checked by using binary search. We observe that most patterns have bound predicates, so these vertices are rarely used for anything but to check for the existence of a triple. We also observe that inserts into this array have $O(n)$ worst case

time complexity, where n is the highest number of predicates that connect a subject with an object. In practice, this was not an issue, since there are usually few predicates for a given subject/object pair.

4 Preliminary Analysis

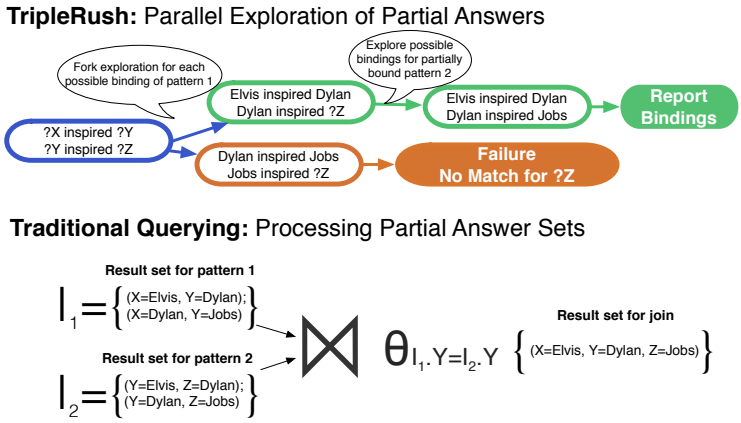


Fig. 3. Comparison between query set processing and TripleRush parallel asynchronous partial answer exploration. Same query and data as in Fig. 2.

As introduced in the last section and illustrated in Figure 3, d-TripleRush processes SPARQL queries by exploring each partial binding asynchronously in parallel. Whenever an exploration encounters more than one possible partial binding, it forks the exploration and pursues both potential solutions in parallel ('green' and 'orange' explorations in the figure). When all variables of an exploration are bound, it returns a result ('green' path). Alternatively, when the remaining unbound variables of an exploration cannot be bound, it aborts that path ('orange' path). Essentially, d-TripleRush performs a parallel-asynchronous graph search.

Traditional DBMS use operators on indices and intermediate data structures (typically arrays or sets). Originally, these operators were executed synchronously, where each operator is executed until its full result set is available before the next operator is called (see also Figure 3). Parallelism is usually introduced by (i) executing independent operators in

parallel (such as the scans that create the sets in the figure) and (ii) implementing parallelised operators resulting in a parallel but synchronous system (each operator has to find all its results before invoking the next one). Modern systems introduce additional parallelism via pipelining operators [5], which allow some operators to pass on partial results. Conceptually, these systems use parallelised approaches to process partial answer sets rather than exploring all possible partial solutions in parallel.

The central proposition of this paper is that d-TripleRush's parallel-asynchronous exploration approach may be a viable alternative graph store architecture for today's multi-core systems.

First, we believe that asynchronous-parallel query processing allows to exploit the many cores better than synchronous-parallel execution, as cores are less likely to wait for work during synchronisation. d-TripleRush is built to exploit this asynchronicity. As mentioned, some current systems exploit a kind of asynchronicity via pipelining. Pipelining, however, comes at the cost of more complexity in both the operators and their coordination. Given that d-TripleRush does not require coordinating between its explorations, it does not incur such an overhead.

Second, we expect the exploitable performance improvement due to parallelism to be curbed by (i) the branching factor of the query, which is a function of the selectivity of the triple patterns and connectivity of the involved nodes (or join selectivity), as it limits the degree of parallelism and (ii) possible gains through locality, as forking explorations and moving them to other cores (possibly on other machines) can be costly operations.

d-TripleRush's index can be conceptualised as a vertical partition of the data into S-Index, P-Index, and O-Index, for subjects, predicates and objects, respectively, as well as three additional indices for each combination of two columns – SP-Index, PO-Index and SO-Index. In addition, the latter three indices are sharded by the subject key (for SP-Index and SO-Index) or object key (for PO-Index). Each shard is assigned to a processing unit in a distributed compute cluster.

5 Evaluation

The goal of the evaluation was to explore the propositions that that d-TripleRush's parallel-asynchronous exploration approach is both competitive and scalable via the efficient exploitation of parallelism where

possible. To that end we employ two standard benchmarks—LUBM and BSBM—and evaluate d-TripleRush's performance under different conditions.
All experiments were run on a cluster of 8 machines, each machine having 128 GB RAM and two E5-2680 v2 at 2.80GHz processors, with 10 cores per processor. The machines are connected with 40Gbps Infiniband. We used version 1.8.0_05-b13 of the Java Runtime.

We used both the LUBM[3] (Lehigh University Benchmark) and BSBM (Berlin SPARQL) [2] benchmarks. For LUBM, we used the queries used in the Trinity.RDF evaluation [19]. For BSBM, we generated the datasets and explore use case queries with the standard data generator and query test driver, but stripped the queries of advanced SPARQL features unsupported by d-TripleRush such as OPTIONAL or complex filters, and discarded queries 9 and 12 for relying on such features.

We executed ten runs of each LUBM query and in the diagrams report both the average and geometric mean over the fastest runs. For BSBM we executed the same ten generated queries from each category, computed the category average and report the average and geometric mean over all categories. The measured total time for a run includes everything from query optimisation until the result set is fully traversed, but the decoding of the results is not forced.

5.1 Vertical scalability and first results

The goal of this evaluation was to measure how well d-TripleRush scales with additional worker threads on a single machine of the cluster. Additionally we measured the time until the first result is reported in order to test our hypothesis that the fully asynchronous execution allows to deliver the first result much faster than the full result set. We ran this evaluation ten times on the LUBM 160 dataset with the Trinity.RDF queries and varied the number of worker threads between 1 and 20, because the hardware has 20 physical cores. We pre-planned the queries and ran them without the optimiser in order to reduce overhead that is not directly associated with the execution engine.

In Figure 4 we see that adding more workers has a negative and at best neutral impact for queries L4, L5, and L6, which touch very little data and are answered in at most a millisecond. For queries L1, L3, and L7,

[3] http://swat.cse.lehigh.edu/projects/lubm

which are more processing intense, the speedup for 20 workers relative to 1 worker is between 10 and 12, which is good, considering that query dispatch and result reporting is still handled by only one worker, and that all queries are answered in under 50ms at that point. Query 2 scales a bit up to 10 worker threads, but does not improve with more processing elements. This is likely due to its structure of only 2 triple-patterns, which offers d-TripleRush less potential for parallelisation.

Figure 4(c) graphs time until the first result was reported relative to the total query execution time (Query 3 was omitted as it does not return results). For queries that profited from parallelisation the first answer was delivered in around a third of the time it took to compute the entire result. The relative benefit increased when going from 1 to 10 worker threads, but then remains approximately constant when going to 20 processing threads.

Overall this evaluation shows that the architecture can take advantage of multicore architectures and that if there are enough workers available, then, for some queries, the asynchronous-parallel execution can deliver first results much sooner than the full results.

5.2 Data scalability and memory usage

To measure the data scalability of d-TripleRush in the single-machine setup we measured its performance for different sizes of the benchmark datasets. For comparison, we also supply the numbers for the in-memory backend of Sesame, as it is also open-source and runs in the JVM, and for Virtuoso 7.1 as a comparison to on-disk approaches.

To make the comparison with the on-disk system Virtuoso fairer, we evaluated warm-cache runs and we configured it to make use of the processors and memory of the machine.

The two diagrams in Figure 5 show how the performance changes, when the LUBM and BSBM queries are executed on increasingly large datasets. On the BSBM dataset the performance of all systems is comparable for small dataset sizes, but d-TripleRush scales better to large dataset sizes, for the largest BSBM dataset it is on average up to 10 times faster than Sesame and up to 25 times faster than Virtuoso. The geometric mean does not change dramatically, because most queries do not touch more data on a larger dataset.

On the more processing intense LUBM queries d-TripleRush shows better performance on any dataset size, up to more than 200 times faster for Sesame and 35 times faster for Virtuoso on average for the largest evaluated size.

We do not have any precise memory measurements, but we measured the used JVM memory, which can serve as an upper bound for the memory used by the index. We then look at the lowest such upper bound that we measured during any of the runs. For BSBM 284'826 d-TripleRush had a lowest upper bound of 39.7 GB, in contrast to 28.6 GB for Sesame. For LUBM 1280 d-TripleRush used 61.6 GB compared to the 34.7 GB used by Sesame. From this we conclude that the d-TripleRush index most likely uses more memory than Sesame's, but that the index size is still reasonable.

5.3 Horizontal scalability

The goal of this evaluation was to measure RW-TR's scalability in the distributed setting. In particular, we wanted to explore if RW-TR's query evaluation approach would degrade when faced with messaging over the network rather than in-memory, or if the benefit of additional processors would dominate. For this, we measured the performance on large BSBM and LUBM data sets while varying the number of nodes used.

Figure 6 shows the results of these evaluations. We aggregated over the fastest runs of ten executions for each query, in order to reduce confounding factors (e.g. garbage collections). We found that for the BSBM dataset/queries the average execution time stays approximately the same, while the geometric mean slightly increases. For the LUBM dataset/-queries the geometric mean stays approximately the same, whilst the average execution time decreases. Our interpretation is that for queries that do not require a lot of processing the added overhead and network latency reduces the performance, whilst for queries that require a lot of processing the benefit of the added processing elements can overcome this drawback. This explains why adding nodes tends to slow down the execution of the fastest millisecond-range queries, whilst improving the performance for the most processing-intense queries.

5.4 Comparison with Trinity.RDF and TriAD

Tables 1 and 2 compare the performance of d-TripleRush to the numbers reported in the Trinity.RDF [19] and TriAD [5] papers. We followed the evaluation procedure described to us by the Trinity.RDF authors, which includes a partitioning of rdf:type into a different type predicate for each class referred to as the object. The d-TripleRush distributed evaluation on LUBM 10240 with 1.36 billion triples was run on all 8 nodes of the cluster. The comparison of the numbers in these tables has many caveats, as the cited numbers were created with different hardware and cluster sizes and the approaches require different amounts of preprocessing. We believe this comparison at least shows that the d-TripleRush architecture is competitive in both the single-node and in the distributed scenario.

Fastest of 10 runs	L1	L2	L3	L4	L5	L6	L7	Geo. mean
TripleRush	22.6	27.8	0.4	1	0.4	0.9	21.2	2.94
Trinity.RDF	281	132	110	5	4	9	630	46
TriAD	427	117	210	2	0.5	19	693	39
TriAD-SG	97	140	31	1	0.2	1.8	711	14

Table 1. Single-node, LUBM 160 (∼21 million triples), time in ms. Comparison data from [19] and [5].

Fastest of 10 runs	L1	L2	L3	L4	L5	L6	L7	Geo. mean
TripleRush	3,111.2	1,457.9	0.7	3.5	9.5	29.1	1,165.8	62.1
Trinity.RDF	12,648	6,018	8,735	5	4	9	31,214	450
TriAD	7,631	1,663	4,290	2.1	0.5	69	14,895	249
TriAD-SG	2,146	2,025	1,647	1.3	0.7	1.4	16,863	106

Table 2. Distributed, LUBM 10240 (∼1.36 billion triples), time in ms. Comparison data from [19] and [5].

6 Limitations

In the following we discuss limitations and threats to validity, followed by the conclusion.

There are some limitations related to the d-TripleRush implementation being a prototype: (i) Encoded IDs cannot exceed 2^{31}, (ii) Only a

subset of SPARQL is supported, (iii) Dictionary encoding/decoding is not distributed, and (iv) Splay integer sets do not currently support deletions. These limitations are not inherent to the approach and resolving them is primarily a matter of engineering.

There are, however, limitations that are inherent to the approach: First, some operations, such as ordering the results, by definition require a synchronisation. Our current approach can only handle them as post-processing steps, which is straightforward but inefficient. Second, an efficient execution of filters requires for an optimiser to be able to place them at any point during the query execution plan. Our current approach is limited to treating them as a post-processing step. More efficient handling would need the ability to access literals from inside the store. This would require an extension of the approach and architecture.

An important limitation of the evaluation is the threat to external validity due to only benchmarking d-TripleRush on synthetic datasets.

This paper focuses on the general architectural approach of d-TripleRush. Components such as the Splay tree and query optimiser are not described, however, they are available as part of the source code.

7 Conclusions

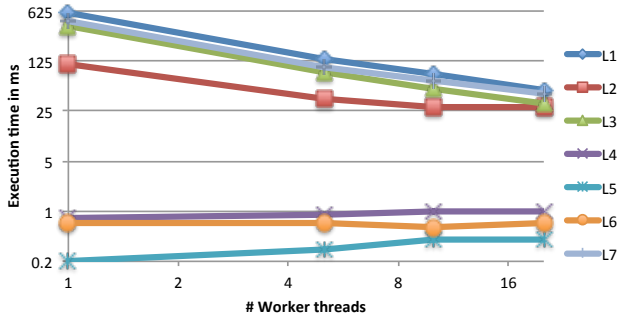
The need for efficient querying of large graphs lies at the heart of most Semantic Web applications. The last decade of research in this area has shown tremendous progress based on database inspired paradigms.

In this paper we proposed to exploit the large number of execution units of modern servers via the parallel exploration of partial bindings layered on top of a distributed graph processing middleware. In particular, we suggested to search for partial bindings to a query as a parallel graph exploration, forking the execution whenever more than one binding is possible, returning the result when all variables of an exploration are bound, and expiring the exploration when it reaches a dead-end.

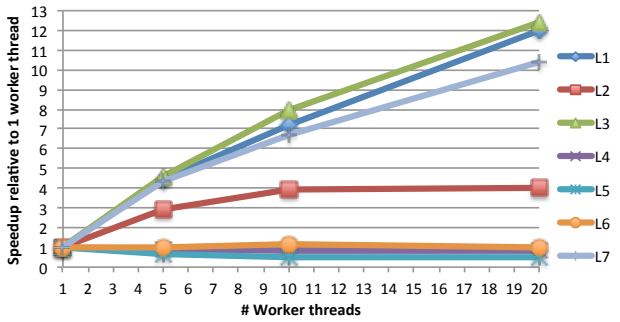
Our evaluation shows that this architecture can serve as the basis for a graph store that is competitive with other systems and offers unique trade offs that allow it to, for example, deliver first results quickly.

Whilst TripleRush has its limitations, it is a step towards building high-performance triple stores that are designed to embrace parallelism.

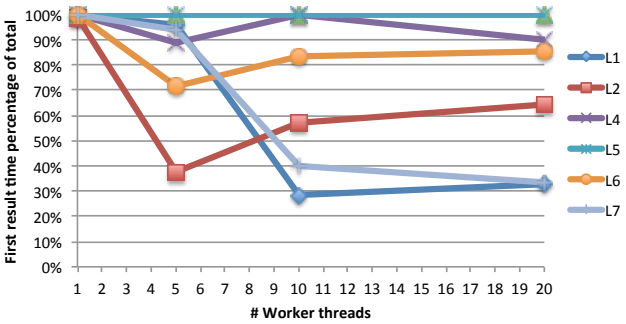
Acknowledgements We thank the Hasler Foundation for supporting this research and Lorenz Fischer for his input on earlier versions of the project.



(a)

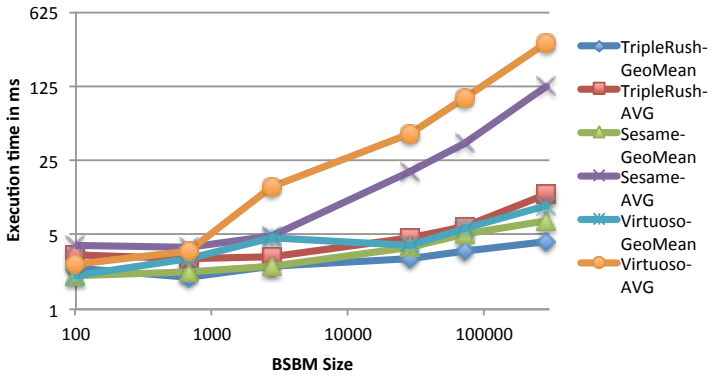


(b)

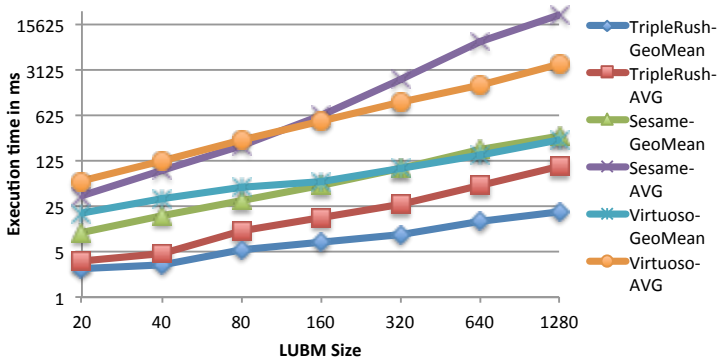


(c)

Fig. 4. 4(a) shows the execution times of the different queries on a logarithmic scale on both axes, 4(b) shows the speedup relative to 1 worker thread for all queries, and 4(c) shows the time it took until the first result as a percentage of the total for the entire result.

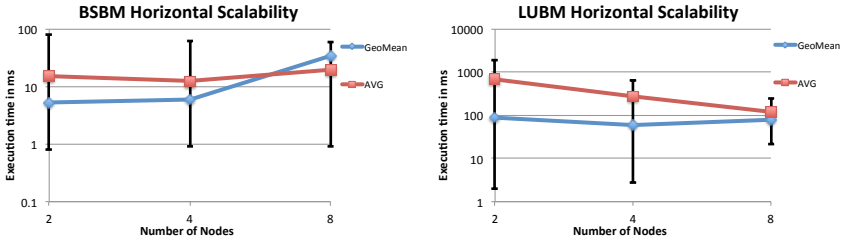


(a)



(b)

Fig. 5. 5(a) and 5(b) compare the single-node scalability of execution times with increasing BSBM and LUBM sizes. Both axes are logarithmic.



(a) (b)

Fig. 6. 6(a) and 6(b) compare the horizontal scalability of RW-TR with 2, 4, and 8 nodes for both BSBM 284'826 and for LUBM 1280. The aggregates are over all queries for that dataset, and for each query we used the fastest of 10 runs. Error bars indicate the runtimes for the fastest and slowest queries in the benchmark.

References

[1] D.J. Abadi, A. Marcus, S.R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 411–422. VLDB Endowment, 2007.

[2] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1): 107–113, 2008.

[4] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, Berkeley, CA, USA, 2012. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387883`.

[5] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 289–300, 2014.

[6] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

[7] Spyros Kotoulas, Jacopo Urbani, Peter A. Boncz, and Peter Mika. Robust runtime optimization and skew-resistant execution of analytical sparql queries on pig. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7649 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2012. ISBN 978-3-642-35175-4.

[8] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[9] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *SIGMOD*. ACM, 2010. ISBN 978-1-4503-0032-2.

[10] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal.The International Journal on Very Large Data Bases*, 19(1):91–113, 2010.

[11] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 627–640, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559911. URL `http://doi.acm.org/10.1145/1559845.1559911`.

[12] Bin Shao, Haixun Wang, and Yatao Li. The trinity graph engine. Technical report, Technical Report 161291, Microsoft Research, 2012.

[13] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, pages 595–604. ACM, 2008.

[14] Philip Stutz, Abraham Bernstein, and William W. Cohen. Signal/-Collect: Graph Algorithms for the (Semantic) Web. In *ISWC*, 2010.

[15] Philip Stutz, Mihaela Verman, Lorenz Fischer, and Abraham Bernstein. Triplerush: A fast and scalable triple store. In *9th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, volume 50, 2013.

[16] Philip Stutz, Bibek Paudel, Mihaela Verman, and Abraham Bernstein. Random walk triplerush: Asynchronous graph querying and sampling. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1034–1044, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-3469-3. doi: 10.1145/2736277.2741687. URL `https://doi.org/10.1145/2736277.2741687`.

[17] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[18] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[19] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. *Proceedings of the VLDB Endowment*, 6(4), 2013.

foxPSL

A Fast, Optimized and eXtended PSL Implementation

This chapter is based on an article published in the International Journal of Approximate Reasoning [10],[4] which is in turn a significant extension of the ideas described in a submission to the KRR 2015 symposium.[5] The co-authors have kindly agreed to permit inclusion as part of this thesis.

[4] http://dx.doi.org/10.1016/j.ijar.2015.05.012, published under a Creative Commons license: https://creativecommons.org/licenses/by/4.0/. Compared to this version the formatting was adapted and there were minor edits to improve consistency and clarity.

[5] https://sites.google.com/site/krr2015/home/accepted-papers

foxPSL

A Fast, Optimized and eXtended PSL Implementation

Sara Magliacane[1], Philip Stutz[2], Paul Groth[1], and Abraham Bernstein[2]

[1] VU University Amsterdam, Amsterdam, The Netherlands
{s.magliacane, p.t.groth}@vu.nl
[2] University of Zurich, Zurich, Switzerland
philip@stutz.tech, bernstein@ifi.uzh.ch

Abstract. In this paper, we describe *foxPSL*, a fast, optimized and extended implementation of Probabilistic Soft Logic (PSL) based on the distributed graph processing framework SIGNAL/COLLECT. PSL is one of the leading formalisms of statistical relational learning, a recently developed field of machine learning that aims at representing both uncertainty and rich relational structures, usually by combining logical representations with probabilistic graphical models. PSL can be seen as both a probabilistic logic and a template language for hinge-loss Markov random fields (MRF), a type of continuous MRF in which maximum a posteriori (MAP) inference is very efficient, since it can be formulated as a constrained convex minimization problem, as opposed to a discrete optimization problem for standard MRFs. From the logical perspective, a key feature of PSL is the capability to represent soft truth values, allowing the expression of complex domain knowledge, like degrees of truth, in parallel with uncertainty.
foxPSL supports the full PSL pipeline from problem definition to a distributed solver that implements the alternating direction method of multipliers (ADMM) consensus optimization. It provides a domain-specific language (DSL) that extends standard PSL with a class system and existential quantifiers, allowing for efficient grounding. Moreover, it implements a series of configurable optimizations, like optimized grounding of constraints and lazy inference, that improve grounding and inference time.
We perform an extensive evaluation, comparing the performance of *foxPSL* to a state-of-the-art implementation of ADMM consensus optimization in GraphLab, and show an improvement in both inference time and solution quality. Moreover, we evaluate the impact of the optimizations on the execution time and discuss the trade-offs related to each optimization.

Keywords: PSL, ADMM, large-scale graph processing

1 Introduction

Probabilistic soft logic (PSL) [4, 7, 1, 2] is one of the leading formalisms of statistical relational learning, a recently developed field of machine

learning that aims at representing both uncertainty and rich relational structures, usually by combining logical representations with probabilistic graphical models. Statistical relational learning has been successfully applied in collective classification, link prediction and property prediction, in a variety of domains from social network analysis to knowledge base construction [6].

PSL can be seen as both a probabilistic logic and a template language for hinge-loss Markov random fields (MRF), a type of continuous MRF with hinge-loss potentials. Similar to other statistical relational learning formalisms such as Markov logic networks (MLN), the first order logic formulae representing the templates can be instantiated (grounded) using the individuals in the domain, creating a MRF on which we can perform inference tasks. A key feature of PSL is the capability to represent and combine soft truth values, i.e., truth values in the interval [0,1], allowing the expression of degrees of truth within complex domain knowledge, in parallel with the uncertainty represented by the probabilistic rules. Given the continuous nature of the truth values, the use of Lukasiewicz operators, and the restriction of logical formulae to Horn clauses with disjunctive heads, maximum a posteriori (MAP) inference in PSL can be formulated as a constrained convex minimization problem. This problem can be cast as a consensus optimization problem [1, 2] and solved efficiently with distributed algorithms such as the alternating direction method of multipliers (ADMM), recently popularized by [3].

The current reference implementation of PSL, described in [1, 2], is limited to running on one machine, which constrains the solvable problem sizes. In [11], PSL is used as a motivation for the development of ACO, a vertex programming algorithm for ADMM consensus optimization that works in a distributed environment. In that work, GraphLab [8] is used as the underlying processing framework.

In this paper we extend the work initially presented in a workshop paper [9]. Specifically, we introduce *foxPSL*,[3] to the best of our knowledge the first end-to-end distributed implementation of PSL that provides an environment for working with large PSL domains, and demonstrate an improvement on the ACO system. Like [11], we adopted a distributed graph processing framework for the basis of our implementation. Instead of GraphLab, we implemented ADMM consensus optimization in SIG-

[3] Apache 2.0 licensed, `https://github.com/uzh/fox`

NAL/COLLECT [12]. Furthermore, we provided a domain specific language (DSL) that extends PSL with a class system, partially grounded rules and existential quantification.

On top of the contributions sketched in the workshop paper, we introduce a series of optimizations to *foxPSL* and improve the preliminary evaluation presented in [9] by an extensive evaluation, which includes exploring two use cases and investigating the impact of the newly introduced optimizations. In the following, we describe *foxPSL* and its features, present an empirical evaluation, and conclude with a discussion of future work.

2 The foxPSL language

As input to *foxPSL* a user can describe the problem in terms of the *foxPSL* language, an intuitive DSL for extended PSL. A problem description consists of a definition of individuals, predicates, facts and rules. In the following, we comment some lines of an example that one can find in the repository.[4]

2.1 Individuals and classes

In *foxPSL* we can explicitly list individuals in the domain, and optionally assign them to a class. For example:

```
class Person: anna, bob
class Party: demo, repub
class LivingBeing: kitty, anna, bob
class Course: ai, db
individuals: ufo
```

By convention individuals always start with a lower-case letter. This distinguishes them from variables, which always start with an upper-case letter. Our domain consists of eight individuals: two individuals of class Person ($anna, bob$), two of class Party ($demo$, $repub$), three of class LivingBeing ($anna, bob$, $kitty$), two of class Course (ai, db) and one individual without any class (ufo). Classes are not mutually exclusive and the

[4] https://github.com/uzh/fox/blob/master/examples/feature-complete.psl

same individual can have multiple classes ($anna$ and bob are both members of LivingBeing and Person). In addition to explicit class assignment, *foxPSL* automatically infers individuals and their classes from facts.

2.2 Predicates and predicate properties

For each predicate, we can optionally specify the classes of its arguments:

```
predicate: professor(_)
predicate: teaches(Person, Course, Person)
```

In the example, the predicate $professor$ takes an argument of any class, while $teaches$ takes a first argument of class Person, a second of class Course and a third of class Person. As we will see in Section 3, this represents a useful hint for the grounding phase, in which first order formulae are grounded (instantiated) with all the possible individuals in the domain. Using the class information, the only individuals that will be used to ground $teaches$ will be of class Person for the first and third argument (i.e. $anna$, bob) and of class Course for the second argument (i.e. ai, db), greatly reducing the number of grounded predicates produced with respect to a class-less predicate.

As in standard PSL, we can define predicate properties such as functionality, partial functionality and symmetry. These properties are translated into constraints on grounded predicates during the grounding.

```
predicate [Functional]: votes(Person, Party)
predicate [Symmetric]: friends(Person, Person)
```

The functional property of $votes$ means that the votes for different parties that a certain person can cast must sum up to 1. The symmetry of $married$ means that for all individuals a, b, if $married(a, b) = x$ then $married(b, a) = x$.

Additionally, we can define a prior for each predicate, representing the starting truth value of a grounded predicate.

```
predicate [prior = 0.5]: young(LivingBeing)
predicate [prior = 0.0]: retired(Person)
```

In the example, a certain grounded predicate of predicate $young$, e.g. $young(kitty)$, will be assigned a truth value of 0.5 if there is no evidence in the knowledge base for another value. A grounded predicate of $retired$

will be assigned a starting truth value of 0.0, which is also the default in *foxPSL*.

2.3 Facts and implicitly defined individuals

Once we have defined the predicates, we can state some facts about our domain and their truth values. If the truth value is not mentioned, we consider it to be 1.

```
fact: professor(bob)
fact [truthValue = 0.8]: friends(anna, bob)
fact [0.9]: !votes(anna, greenparty)
```

In our domain, bob is a professor with truth value 1 and $anna$ is a friend of bob with truth value 0.8. Since $friends$ is a symmetric predicate, it means that we expect $friends(bob, anna)$ to be true with the same truth value. Moreover, $anna$ does not (negation !) vote for $greenparty$ with truth value 0.9 (we can omit $truthValue$), which means $votes(anna, greenparty) = 0.1$. Although $greenparty$ was not mentioned as a Party before, it will be inferred as such because it is the second argument in a $votes$ fact.

2.4 Rules, existential quantifiers and partial groundings

The core of *foxPSL* is the definition of rules, which are Horn rules with disjunctive heads. The restriction to this form is a constraint of standard PSL that enables the translation to convex functions. In the following example, the upper-case names represent the variables in the rules.

```
rule [weight=5]: votes(A,P) & friends(A,B) => votes(B,P)
rule [3]: young(L) => !retired(L)
```

Similar to standard PSL, each rule can have an associated weight that represents how much it would cost to be violated. If the weight is not specified, we consider the rule a hard rule, therefore a rule that must be always true in the domain. As an implementation choice, we ground each variable in a rule only to individuals that are part of all of the classes of the predicate arguments in which the variable appears. For example, in the second rule L is only grounded with individuals in the intersection of the classes LivingBeing and Person, because it appears in both $young$ which requires a LivingBeing and $retired$ which requires a Person.

We allow rules that are partially grounded, i.e., that contain both individuals and variables:

```
rule [weight=3]: member(A, wwf) => votes(A, greenparty)
```

Here wwf and $greenparty$ are both individuals (since they start with a lower case letter), while A is a normal variable.

In addition to standard PSL, we introduce the existential quantifier $EXISTS[variables]$, which can only appear in the head, in order to preserve convexity. Given the finite domain and class system, we can safely substitute the existential quantifiers by unrolling them to disjunctions.

```
rule: professor(P) => EXISTS [C,S] teaches(P,C,S) | retired(P)
```

Similarly to other variables in the rule, the existential quantifier is grounded using the intersection of the classes of the predicate arguments where the variable appears. For example C is grounded to any individual of class Course, while S is grounded to any individual of class Person. In our current domain, the above rule can be safely rewritten as:

```
rule: professor(P) => teaches(P,ai,bob) | teaches(P,ai,anna) |
      teaches(P,db,bob) | teaches(P,db,anna) | retired(P)
```

3 System Description

foxPSL is designed as a pipeline, consisting of a grounding, graph construction and inference/consensus optimization phase, similarly to most PSL implementations. The grounding phase is centralized, while the graph construction and inference are implemented in the SIGNAL/COLLECT graph processing framework. In this section we describe the underlying graph processing framework and each stage of the pipeline (see Figure 1) with its inputs and outputs.

3.1 Graph processing in Signal/Collect

SIGNAL/COLLECT [12][5] is a parallel and distributed graph processing framework. Akin to GraphLab [8], it allows the formulation of computations in terms of vertex centric methods. In SIGNAL/COLLECT functions are separated into ones for aggregating received messages (collecting)

[5] http://uzh.github.io/signal-collect/

and for computing the messages that are sent along edges (signaling). In contrast to GraphLab, SIGNAL/COLLECT supports different vertex types for different processing tasks, which allows for the natural representation of bipartite ADMM consensus optimization graphs by using two vertex types. SIGNAL/COLLECT also supports configurable convergence detection that can be based both on local and global properties. The global convergence detection is based on efficient MapReduce-style aggregations. Additional features such as dynamic graph modifications and vertex placement heuristics to reduce communication over the network are supported and might enable future *foxPSL* extensions (see Section 6).

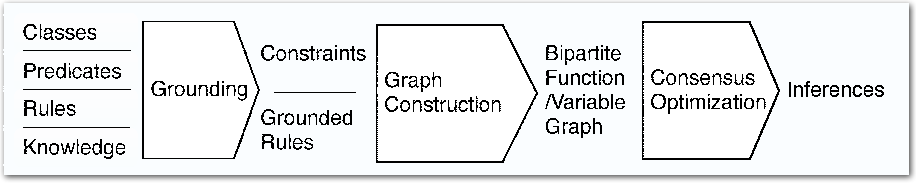


Fig. 1. The architecture of the *foxPSL* system is a pipeline that takes the rules, instance knowledge and metadata as inputs and through a series of processing steps computes the inferences.

3.2 Grounding

Grounding is well studied in a number of communities, especially in logic programming. In most probabilistic logics based on probabilistic graphical models (MLN, PSL, etc.), first order logic formulae are instantiated (grounded) using the individuals in the domain, creating a propositional representation that is used to generate a graphical model.

Given the *foxPSL* class system and language extensions, the grounding procedure is different from the mentioned work. For each predicate that is mentioned in a rule we instantiate each argument with all suitable individuals, creating several grounded predicates from a single predicate.

For example, for the rule:

```
rule [weight=5]: votes(A,P) & friends(A,B) => votes(B,P)
```

the predicate $votes(A, P)$ produces 6 grounded predicates: $votes(anna, demo)$, $votes(anna, repub)$, $votes(anna, greenparty)$, $votes(bob, demo)$, $votes(bob, repub)$, $votes(bob, greenparty)$.

In this phase we leverage class information of *foxPSL* to reduce the number of groundings from any individual to only the ones that match the classes of arguments. Some of these grounded predicates are user provided facts with truth values (e.g. $votes(anna, greenparty) = 0.1$), but most of them have an unknown truth value that we will compute.

Substituting the grounded predicates in all combinations in each rule, we can generate several grounded rules. The above rule generates 18 grounded rules, e.g.,

$$weight = 5 : votes(anna, demo) \& friends(anna, bob) => votes(bob, demo) \quad (1)$$

The same substitution is done for all constraints. For example, the functional property of $votes$ gets grounded to 2 functional constraints, e.g. for $anna$: $votes(anna, demo) + votes(anna, repub) + votes(anna, greenparty) = 1$.

3.3 Graph construction

The grounding step produces a set of grounded rules and constraints, each containing multiple instances of grounded predicates. As defined in [4, 1], each grounded rule is translated to a potential of a hinge-loss Markov random field (HL-MRF), a type of continuous Markov random field with constraints, using Lukasiewicz operators:

$$x \wedge y = max(0, x + y - 1)$$

$$x \vee y = min(1, x + y)$$

$$\neg x = 1 - x$$

Using these operators on a grounded rule in the form:

$$[weight]\ b_1 \wedge ... \wedge b_n => h_1 \vee ... \vee h_m \quad (2)$$

the authors [4, 1] define the distance to satisfaction as:

$$max(0, b_1 + ... + b_n - n + 1 - h_1 - ... - h_m)^p \quad (3)$$

where $p = 1$ for rules with linear distance measures and $p = 2$ for rules with squared distance measures. The intuition is that a squared distance

measure is more forgiving when the violation is small and more penalizing when it is large. In *foxPSL* we can define for each rule its own distance measure at the moment of its declaration by specifying the parameter $distanceMeasure$. If nothing is specified, the default distance measure is squared. For example, the grounded rule from the above subsection, becomes:

$$max(0, votes(anna, demo) + friends(anna, bob) - 1 - votes(bob, demo))^2 \tag{4}$$

The weighted sum of the potentials defines the energy function of the HL-MRF with a domain constrained by the grounded constraints. As in standard MRFs, the probability is defined as an exponential of the energy function, therefore finding the assignment of unknown variables that maximizes the probability of a certain world (maximum a posteriori or MAP inference) is equivalent to minimizing the energy function.

Since the potentials are by construction convex functions and the constraints are defined to be linear functions, MAP inference is a constrained convex minimization problem. Following the approach described in [1, 2, 11] we solve the MAP inference using consensus optimization. We represent the problem as a bipartite graph, where grounded rules and constraints are represented with function (also called subproblem) vertices and grounded predicates are represented with consensus variable vertices. Each function (grounded rule or constraint) has a bidirectional edge pointing to every consensus variable (grounded predicate) it contains.

3.4 Inference: consensus optimization with ADMM

The function/consensus variable vertices implement the alternating direction method of multipliers (ADMM) consensus optimization [3]. Intuitively, in each iteration each function is minimized separately based on the consensus assignment from the last iteration and the related Lagrange multipliers. Once done, the function vertex sends a message with the preferred variable assignment to all connected consensus variables. Each consensus variable computes a new consensus assignment based on the values from all connected functions by averaging the received values, and sends it back. This process is repeated until convergence, or in the approximate case, until the primal and dual residuals fall below a

threshold based on the parameters $\epsilon_{abs}, \epsilon_{rel}$. These stopping criteria are also recommended and described in [3]. Since each variable represents a grounded predicate, the assignment to a variable is the inferred soft truth value for that grounded predicate. At the end of the computation the system returns all inferred truth values.

3.5 Termination: convergence criteria comparison with ACO

As described above, *foxPSL* stops when the total primal and dual residuals are below the threshold defined in [3]. This is detected by using SIGNAL/COLLECT global convergence detection, which is configured to compute the global primal and dual residuals after every iteration and compare them with the thresholds. The residual computation is implemented as a MapReduce-style aggregation over the vertices.

The implementers of ACO [11] argue that such global aggregations are wasteful, because the residual contributions of many parts of a graph often do not change much after a few iterations. For this reason they instead implement a local convergence detection algorithm in which each consensus variable computes a local approximation of the residuals using its local copies. If the local residuals in a consensus variable are smaller than a local threshold, it does not communicate to the related subproblem. If the subproblem is not awakened by any of its consensus variables, then it isn't scheduled again until there is a change in the consensus variables. The global computation stops once all local computations have converged. The disadvantage of this approach is that certain local computations may converge too eagerly, requiring others, possibly involved in more complex negotiations, to run for more iterations in order to achieve overall residuals that are below a certain global threshold. We suspect that this is the main reason for the difference in solution quality and convergence between ACO and *foxPSL* in the experiments in Section 5.

4 Optimizations

foxPSL implements a series of configurable optimizations for each phase of the system. As we discuss in Section 5, the optimizations do not improve the inference time on all settings and some may require tuning to be effective.

4.1 Grounding and graph construction phases: optimizations on constraints

We implement three optimizations on the constraints: removing symmetric constraints, removing trivial functional constraints and pushing trivial partial functional constraints to bounds on the consensus variables.

Symmetric constraints are produced from predicates with the symmetric property: for example, the symmetry of $married$ means that for all individuals a, b, if $married(a, b) = x$ then $married(b, a) = x$. In PSL this is implemented as a constraint in the form: $married(a, b) - married(b, a) = 0$. This optimization simply merges the two grounded predicates and uses $married(a, b)$ in all the instances where a grounded rule would contain $married(b, a)$, reducing the number of constraints and grounded predicates. Counter-intuitively, in the experimental evaluation we show that this simple optimization does not always improve the inference time, although it greatly reduces the grounding time.

Functional constraints are produced from predicates with the functional property, e.g. $votes$. In case only one of the grounded predicates in the grounded constraint, e.g. $votes(anna, demo)$, is not part of the facts and thus not yet bound, we can simply assign its truth value to $1 - votes(anna, repub) + votes(anna, greenparty)$.

Partial functional constraints are similar to functional constraints, with the difference that they require an inequality. In other words, if $votes$ is partial functional, the sum of votes that a voter can cast has to be less than or equal to 1, including the possibility that the voter does not vote. In this case, we cannot assign the value of an unknown grounded predicate, apart from the trivial case in which the truth value is required to be lower than 0. In other cases we can implement the constraint as a bound on the truth value of the consensus variable. This bound will be used in each iteration by the consensus variable to clip the truth value. This optimization improves the one implemented in [11], which uses a similar mechanism to bound the truth value of any consensus variable to the interval [0, 1].

An additional trivial optimization we perform is to remove grounded rules that have a distance to satisfaction of 0 for any possible assignment of the grounded predicates in [0, 1], since they do not influence the minimization.

Algorithm 4 Lazy inference algorithm: subproblem vertex

```
subproblem vertex, state at time t: variables x_1, ..., x_n, Lagrange multipliers y_1, ..., y_n, threshold δ
var sendSelfAwakeningMessage = false;
for i = 1 to n do
    if |x_i^t − x_i^{t−1}| ≥ δ then
        send message x_i^t to consensus variable z_i
    else
        if |y_i^t − y_i^{t−1}| ≥ δ then
            sendSelfAwakeningMessage = true;
        end if
    end if
end for
if sendSelfAwakeningMessage and no other message sent then
    send a message to self to ensure awakening
end if
```

4.2 Inference phase: lazy inference and configurable steps for global convergence detection

We propose two optimizations: lazy inference and a configurable step for global convergence detection.

The lazy inference optimization is similar in spirit to the local convergence detection in [11], allowing for the computation to stop in parts of the graph while it is still active in others. The pseudo-code is shown in Algorithm 4. The main idea is that consensus variables always answer with a message, while each subproblem vertex tries to avoid resending the same message in two successive iterations. If any vertex in SIGNAL/-COLLECT does not receive any message, then it is not scheduled for execution. If a consensus variable is awakened by a message and does not receive any message from a certain subproblem vertex, it will compute the new consensus using an old message for that specific vertex, and send it to all the connected subproblem vertices.

Sending a message only if the message is different from the previous one creates a problem: it is possible that the internal state of the subproblem, the multipliers, are evolving, even if this is not reflected in the messages. If one does not send a message in this case, then that means that the subproblem vertex potentially does not get scheduled again, which could lead to incorrect results. On the other hand, if one sends a message to the consensus vertex whenever only the multiplier changes, then this would always trigger a potentially wasteful rescheduling of *all* subprob-

lem vertices that are connected to that consensus vertex. To overcome this issue we propose the following algorithm: if none of the messages to the connected consensus vertices have changed, but at least one of the multipliers did change, then the subproblem vertex ensures its own rescheduling by sending a special message to itself. We implement this approach with a configurable threshold under which the change is not considered significant. This allows one to choose a trade-off between quick local convergence at the cost of slightly reduced solution quality.

In the standard ADMM algorithm and in ACO [11] convergence detection is performed after each iteration. Since global convergence detection is computationally expensive, there is a trade-off between running it every iteration, thus stopping the computation as soon as possible, and running it every n iterations, thus risking to run the computation for $n-1$ iterations too many. In *foxPSL* we allow for the configuration of convergence detection step size, and in Section 5 we present experiments that illustrate the trade-off.

5 Evaluations

We present an extensive evaluation on synthetic datasets representing two use cases: a voter social network and a movie recommendation use case. In the following, we first describe the datasets and then present the comparison with ACO [11], showing gains in both inference time and solution quality. Secondly, we show the impact of the optimizations on the grounding time and inference time, as well as solution quality, explaining the tradeoffs in the choice of parameters for the optimizations. All evaluations are run on four machines, each with 128 GB RAM and two E5-2680 v2 at 2.80GHz 10-core processors. All machines are connected with 40Gbps Infiniband. *foxPSL* is developed in Scala and runs on version 1.8.0_05-b13 of the Java Runtime.

5.1 Evaluation datasets

We use two sets of synthetically generated datasets, Voter Networks and Movies, each containing four datasets of varying size, to evaluate the performance of our system.

The Voter Networks datasets are the sets of grounded rules used in [11] representing four synthetic social networks of increasing size modeling voter behavior, generated using a synthetic data generator described

in [5]. In Table 1 we show dataset statistics, such as the number of subproblem and consensus vertices, as well as the number of edges connecting them. A more detailed description of the datasets can be found in [11].

Since the Voter Networks datasets contain already grounded rules, we cannot measure the grounding time and the impact of some of our DSL features and optimizations. Therefore, we developed our own synthetic dataset generator. We use a movie recommendation use case, in which we recommend a movie to a user based on the movies, actors and directors that we already know she likes, but also based on what movies, actors and directors the people in her social network and the similar users like. We formulate the problem in the *foxPSL* language, using features of *foxPSL* such as classes, priors, predicate properties and existential quantifiers. As predicates and rules we use:

```
predicate [prior = 0.5]: likes(User, Likable)
predicate [Functional]: directedBy(Movie, Director)
predicate: playsIn(Actor, Movie)
predicate [Symmetric]: friends(User, User)

rule [1000]: EXISTS[MOVIE] directedBy(MOVIE, DIRECTOR)
rule [50]: likes(USER, ACTOR) & playsIn(ACTOR, MOVIE) => likes(USER, MOVIE)
rule [1]: likes(USER, MOVIE) & playsIn(ACTOR, MOVIE) => likes(USER,ACTOR)
rule [100]: likes(USER, DIR) & directedBy(MOVIE, DIR) => likes(USER, MOVIE)
rule [10]: likes(USER, MOVIE) & directedBy(MOVIE, DIR) => likes(USER, DIR)
rule [5]: likes(A, LIKABLE) & friends(A, B) => likes(B, LIKABLE)
rule [0.1]: likes(A, L1) & likes(A, L2) & likes(B, L1) => likes(B, L2)
```

For a given number of individuals, we populate the classes Actor (55.8%), Movie (33%), Director (11.2%) based on the proportions in which they are present in LinkedMDB.[6] All of these individuals constitute also the class Likable. Moreover, we make reasonable assumptions about the class Users, assuming that we have in general 10 times more Users than Likables. Given a proportion of known facts, we create the facts by sampling the individuals that are connected by a predicate using a normal distribution with a certain variance. We also sample the truth value of each fact using a normal distribution with mean 0.5 and standard deviation 0.25.

Using the synthetic dataset generator, we produce four Movies datasets of increasing size, generated using 100, 150, 200 and 250 individuals, a mean proportion of known facts of 0.3 and a standard deviation of 0.1. In Table 1, we show the same statistics as for the Voter Network case

[6] http://wiki.linkedmdb.org/Main/Statistics

also for the Movies datasets. The biggest datasets of each use case are comparable regarding the total number of vertices. In general the Movies datasets have a higher number of edges and a higher proportion of subproblems relative to the number of consensus variables.

Dataset	$\|subproblems\|$	$\|consensusVar\|$	$\|edges\|$
Voter Network 1M	3'307'971	1'102'498	12'129'370
Voter Network 2M	6'656'775	2'204'994	24'422'102
Voter Network 3M	9'962'627	3'307'492	36'543'000
Voter Network 4M	13'349'751	4'409'988	48'989'000
Movies 100	397'465	2'823	2'310'250
Movies 150	2'455'662	6'437	14'502'334
Movies 200	7'956'132	11'740	48'731'492
Movies 250	20'331'433	18'167	121'878'352

Table 1. Dataset comparison. The Social Network datasets are from [11].

5.2 Comparison with ACO

We compare *foxPSL* with ACO, the GraphLab ADMM consensus optimization implementation presented in [11], measuring both the inference time and solution quality. We do this only on the Voter Network datasets, because whilst we can load an already grounded network with ACO, it does not generate a grounding from a PSL-style language. We extended *foxPSL* with a separate loading pipeline for this serialized grounding format.

Both systems run an approximate version of the ADMM algorithm, as described in [3], with the same parameters $\rho = 1$, $\epsilon_{abs} = 10^{-5}$ and $\epsilon_{rel} = 10^{-3}$. As discussed in Section 3, ACO implements a special local convergence detection technique, while *foxPSL* employs the textbook global convergence detection algorithm. We configure both systems to take advantage of the 40 virtual cores on each machine. For *foxPSL* we configure some of the optimizations: we use the lazy optimization with a threshold of 1e-9 and a convergence detection step of 10 iterations. The grounding phase optimizations cannot be applied since the input is already grounded.

Inference time comparison For each of the four datasets, we measure the inference time at a fixed number of iterations for both systems. Figure

2 shows the results averaged over ten runs, limited to 10, 100, and 1000 iterations. Since the ACO implementation performs two extra initial iterations that are not counted in the limit, the comparison is made with an iteration limit of 12, 102, and 1002 for *foxPSL*. We stop the evaluation at 1000 iterations, because *foxPSL* converges in that interval, although ACO does not. Therefore, above the limit of 1000 iterations the running time for *foxPSL* is the same, while it continues to increase for ACO without substantial improvements in quality.

The inference time of *foxPSL* is considerably better for all evaluated iterations, on the two bigger datasets is more than 10 times faster, as shown by the lower computation times (y-axis) in Figures 2. Moreover, the computation time scales linearly with increasing problem size (x-axis).

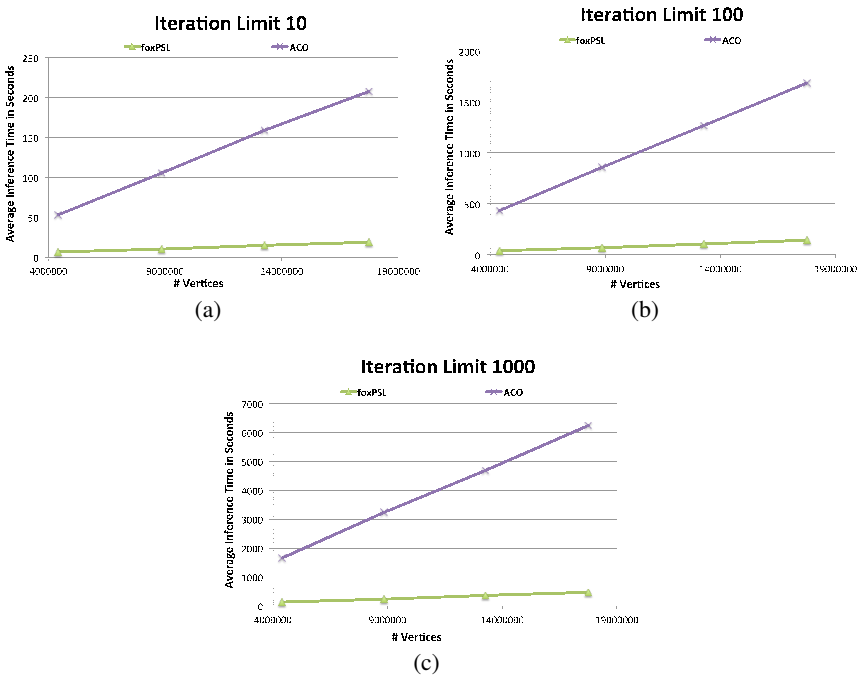


Fig. 2. Figures 2(a), 2(b) and 2(c), compare the average inference time between *foxPSL* and ACO. For each graph size there were ten runs, with an iteration limit of 10, 100 and 1000 respectively.

Solution quality comparison We also compare *foxPSL* and ACO in terms of the solution quality on the same evaluation runs discussed above.

Since PSL inference is a constrained minimization problem solved with an approximate algorithm, we consider two quality measures for solution quality: the objective function value and a measure of the violation of the constraints.

In Table 2, we show a comparison of the solution quality for *foxPSL* and ACO at iteration 1000 from the previous experiments. The four parameters we compare are the objective function value $ObjVal$, the sum of violations of the constraints $\Sigma viol$, the number of violated constraints at tolerance level 0.01 $|viol@0.01|$ and the number of violated constraints at tolerance level 0.1 $|viol@0.1|$. Moreover, we report an estimation of the optimal objective function value $OptVal$ as computed using lower epsilons ($\epsilon_{abs} = 10^{-8}$, $\epsilon_{rel} = 10^{-8}$) which has a sum of constraint violations on the order of only 10^{-5}.

We can see that *foxPSL*'s objective function value is closer to the estimated optimal objective function value than ACO's for all datasets. The sum of the violations of the constraints is lower in *foxPSL* by a factor of more than 10.

		foxPSL				ACO			
$\|vertices\|$	$OptVal$	$ObjVal$	Σv	$\|v@0.01\|$	$\|v@0.1\|$	$ObjVal$	Σv	$\|v@0.01\|$	$\|v@0.1\|$
4410K	4838.14	4809.97	7.0.5	20	4	4722.09	119.55	491	233
8861K	9692.03	9678.66	13.38	18	3	9520.12	233.04	924	431
13.2M	14521.26	14448.14	19.77	42	6	14199.59	349.14	1387	640
17.7M	19425.71	19441.26	26.64	52	5	19119.38	472.49	1894	863

Table 2. Comparison of the solution quality for *foxPSL* and ACO with iteration limit 1000 and same parameters ($\rho = 1$, $\epsilon_{abs} = 10^{-5}$, $\epsilon_{rel} = 10^{-3}$).

Besides the lower total violation of the constraints, Table 2 also shows the number of constraints that are violated by more than a certain threshold of tolerance. In particular, in the ACO solution there are several hundred constraints that are violated even while tolerating errors of 0.1, which is sizable considering that the variables are constrained in the interval [0, 1]. In general, the violations of the ACO solution are larger and less spread across the constraints than the violations found in the *foxPSL* solution, possibly due to the local convergence detection of the

former, which may be too eager to converge on some subgraphs, leading to a solution with a higher approximation error.

5.3 Optimization Evaluation

The goal of the optimization evaluation is twofold: on the one hand we would like to find out how the optimizations affect the performance of the grounding and inference phases of *foxPSL*. On the other hand we would like to answer the question if the impact of the optimization changes with the dataset size. For this reason we first evaluate the performance for *foxPSL* with the baseline settings on datasets of increasing size. For each optimization we then change the feature relative to the baseline configuration in order to gauge the effect on the performance.

We evaluate three optimizations: the symmetric constraint optimization, the global convergence detection step and the lazy inference threshold. The other optimizations on constraints have similar results to the symmetric constraint optimization. We perform the experiments on both use cases with the exception of the symmetric constraint optimization, which cannot be used on the Voter Networks datasets because they contain already grounded rules and they do not contain any symmetric constraints. Our baseline configuration consists of all grounding phase optimizations enabled, lazy inference enabled with a threshold of 10^{-13} and the same parameters as in the ACO comparison, $\rho = 1$, $\epsilon_{abs} = 10^{-5}$ and $\epsilon_{rel} = 10^{-3}$. On the Voter Network dataset we use a convergence detection step of 10, while on the Movies dataset we check convergence at each step.

Symmetric optimization Figure 3 shows the impact of symmetric optimization on the Movies datasets. As already mentioned, we cannot compare on Voter Networks datasets because they contain already grounded rules and they do not contain any symmetric constraint. The optimized grounding and inference times are represented with a dashed line, while the full lines represent the grounding and inference times when the optimization is disabled. For the Movies dataset, shown in Figure 3, the grounding time improves greatly with the optimization with gains increasing with the size of the dataset. On the other hand, the inference time is essentially the same. This is counter-intuitive, since the symmetric optimization reduces the number of vertices, therefore we would expect

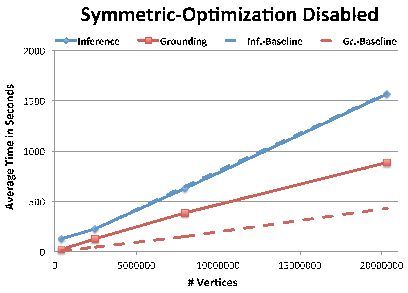
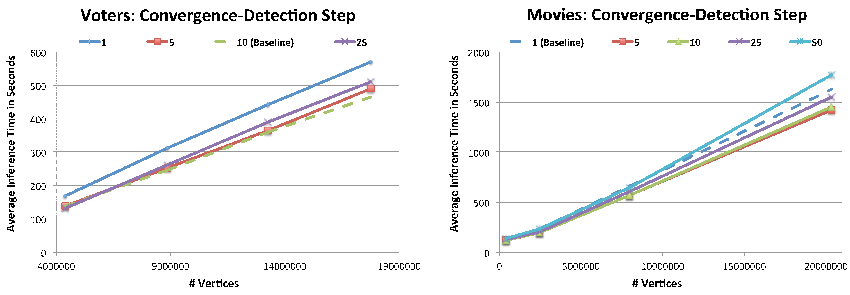


Fig. 3. Symmetric optimization evaluation on Movies datasets: dashed line represents the optimized system.

the inference time to improve. In practice, given the many other rules in this use case, the reduction of vertices is in the order of 0.1%, so the difference is marginal.

Global convergence detection step Figure 4 shows the impact on inference time of running the convergence detection every n iterations on both the Voter Networks and the Movies datasets. In general, there is a trade-off between saving the overhead of a too frequent convergence detection and running more iterations than necessary. In our use cases, the



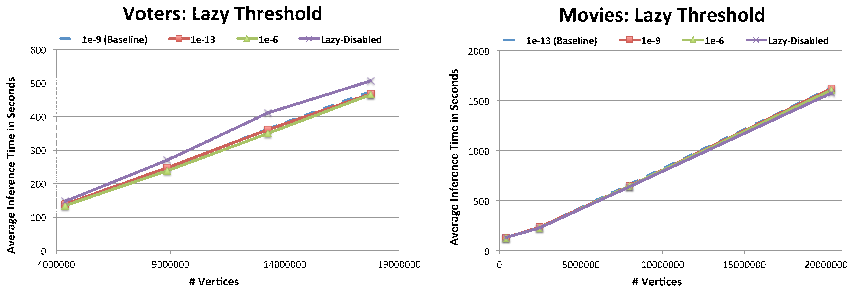
(a) (b)

Fig. 4. Varying the convergence detection step for Voter Networks (a) and Movies (b) datasets.

standard configuration in which we run the convergence detection every step is consistently outperformed by the other configurations. The optimal convergence detection step seems to be between 5 and 10 iterations, decreasing the inference time by 10%, while at 25 iterations the benefits

decrease. Above 25 iterations, the inference time actually worsens with respect to the baseline. We can see that the effects of this optimization increase linearly with the size of the dataset and therefore the cost of computing if global convergence was reached. A positive side effect of a higher convergence step is the slight increase in solution quality, since the computation runs for a few more iterations.

Lazy inference thresholds Figure 5 shows the impact of enabling lazy inference and varying the threshold under which a change is not considered significant on both the Voter Networks and the Movies datasets. On



(a) (b)

Fig. 5. Varying the lazy inference threshold for Voter Networks (a) and Movies (b) datasets.

the Voter Networks datasets, shown in Figure 5(a), enabling lazy inference reduces the inference time by 10%, while the difference between the various thresholds is not clear. In general, a higher threshold, e.g. 10^{-6}, allows for faster inference, but with a slightly lower solution quality (an objective function that is higher by 0.01 and a sum of broken constraints that is higher by 0.05 in the worst case). On the Movies datasets, shown in Figure 5(b), the improvement is very small. We suspect that the difference is due to the homogeneity and the high number of edges of the Movies dataset, which does not allow parts of the graph to converge locally.

6 Limitations and Conclusions

In this paper, we extend the work initially presented in a workshop paper [9] introducing *foxPSL*, to the best of our knowledge the first end-to-

end distributed implementation of PSL that provides an environment for working with large PSL domains, including a domain specific language (DSL) that extends PSL with a class system, partially grounded rules and existential quantification. Moreover, we present a series of optimizations to *foxPSL* and improve the preliminary evaluation presented in [9] by an extensive evaluation, which includes exploring two use cases and investigating the impact of the newly introduced optimizations.

Our current implementation has limitations. Our ADMM algorithm uses a fixed step size, leading to an initially fast approximation of the result and a slow exact convergence. An improvement would be a variant with adaptive step sizes. Additional improvements might be incremental reasoning when facts/rules change, taking advantage of the dynamic graph modification features supported by SIGNAL/COLLECT, and the use of SIGNAL/COLLECT vertex placement heuristics to reduce messaging.

Whilst developing *foxPSL*, we found that PSL provides a powerful formalism for modeling problem domains. However, its power comes with numerous interactions between its elements. Hence, the use of PSL would be aided by a DSL and an end-to-end environment that allows for the systematic analysis of these interactions. We believe that *foxPSL* is a first step towards such an environment.

Acknowledgements We would like to thank Hui Miao for sharing the ACO source code and evaluation datasets, and Stephen Bach for being able to use his optimizer implementations in *foxPSL*, as well as for the useful discussions. Furthermore, we would like to thank the Hasler Foundation and the Dutch national program COMMIT for supporting this work.

References

[1] Stephen H. Bach, Matthias Broecheler, Lise Getoor, and Dianne P. O'Leary. Scaling MPE inference for constrained continuous Markov random fields. In *NIPS*, 2012.

[2] Stephen H. Bach, Bert Huang, Ben London, and Lise Getoor. Hinge-loss Markov random fields: Convex inference for structured prediction. In *UAI*, 2013.

[3] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3(1), January 2011.

[4] Matthias Broecheler, Lilyana Mihalkova, and Lise Getoor. Probabilistic similarity logic. In *UAI*, 2010.

[5] Matthias Broecheler, Paulo Shakarian, and V.S. Subrahmanian. A scalable framework for modeling competitive diffusion in social networks. In *International Conference on Social Computing (SocialCom)*, 2010.

[6] Lise Getoor and Ben Taskar. *Introduction to statistical relational learning*. MIT press, 2007.

[7] Angelika Kimmig, Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *NIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012.

[8] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[9] Sara Magliacane, Philip Stutz, Paul Groth, and Abraham Bernstein. foxpsl: An extended and scalable psl implementation. In *AAAI Spring Symposium on KRR, Knowledge Representation and Reasoning: Integrating Symbolic and Neural Approaches*, 2015.

[10] Sara Magliacane, Philip Stutz, Paul Groth, and Abraham Bernstein. foxpsl: A fast, optimized and extended {PSL} implementation. *International Journal of Approximate Reasoning*, 67:111 – 121, 2015. ISSN 0888-613X. doi: http://dx.doi.org/10.1016/j.ijar

.2015.05.012. URL http://www.sciencedirect.com/science/article/pii/S0888613X15000845.

[11] Hui Miao, XiangYang Liu, B. Huang, and L. Getoor. A hypergraph-partitioned vertex programming approach for large-scale consensus optimization. In *Big Data, 2013 IEEE International Conference on*, 2013.

[12] Philip Stutz, Abraham Bernstein, and William W. Cohen. Signal/-Collect: Graph Algorithms for the (Semantic) Web. In *ISWC*, 2010.

Curriculum Vitae

Curriculum Vitae

Personal Details

Name	Philip Stutz
Place of origin	Embrach, ZH
Citizenship	Swiss

Education

2009 – 2014	Ph.D. Student & Assistant @ University of Zurich Adviser: Professor Abraham Bernstein, Ph.D.
2006 – 2009	MSc ETH in Computer Science, with distinction @ ETH Zurich
2007 – 2008	Erasmus Exchange @ University of Aberdeen in Scotland
2003 – 2008	BSc ETH in Computer Science @ ETH Zurich

Professional Experience

Summer 2012	Software Engineering Intern @ Google Inc. in Mountain View Worked on large-scale graph processing.
2001 – 2003	Allround Internship @ UBS AG in Switzerland